Zigzag Decodable codes: Linear-time erasure codes with applications to data storage

Xueqing Gong *, Chi Wan Sung

A R T I C L E   I N F O

Article history:
Received 6 May 2016
Received in revised form 4 January 2017
Accepted 9 May 2017
Available online 24 May 2017

Keywords:
Zigzag Decodable codes  
k-Reliability  
Erasure codes  
Data storage  
Linear time  
Implementation

A B S T R A C T

An erasure code is said to be k-reliable if it maps k source packets into n coded packets, and any k out of the n coded packets allow recovery of the original k source packets. Codes of k-reliability achieve the best reliability-storage tradeoff, which are useful for fault tolerance in data storage systems. Zigzag Decodable (ZD) codes are k-reliable erasure codes. Its encoding and decoding (per information bit) can be done in linear time, involving only XOR and bit-shift operations. Two classes of ZD codes are constructed, and compared with Cacuhy-RS codes, the state-of-the-art general-purpose MDS codes. Numerical results show that ZD codes outperform Cauchy-RS codes over a wide range of coding parameters.

© 2017 Elsevier Inc. All rights reserved.

1. Introduction

Due to the explosive growth of personal and enterprise digital data, providing good data storage solutions becomes a popular topic. To design a data storage system, high reliability is very important, since no one wants to experience data loss. The simplest way to provide reliability is to replicate the data multiple times and store them into independent storage devices, which can be multiple hard disks or data servers located in geographically separated areas. From the perspective of coding theory, this simple solution corresponds to the use of repetition codes, which is well known to be inefficient in terms of storage space. To improve storage efficiency, one may use erasure codes, which can achieve the same level of reliability with much less redundancy. On the other hand, the use of erasure codes requires the calculation of parity bits when storing a file into the system, and the calculation of information bits when the file is retrieved by assessing storage devices that have stored the parity bits. These operations not only increase the response times of storing and retrieving a file from the storage system, but also consumes more energy due to the computations. As for traditional erasure codes, the encoding and decoding often involve multiplication over a large finite field, which is much more time and energy consuming than addition [1]. For many applications, such as big-data storage systems and mobile systems with hand-held devices that have low computation capability [2], it is important to have a storage system that can provide high accessibility and low energy consumption for both storage and retrieval. Therefore, it is useful to design erasure codes that have low encoding and decoding complexities.

Erasure codes have been adopted in data storage systems such as Oceanstore, Total Recall systems, and Windows Azure Storage. The most important group of erasure codes are Maximum Distance Separable (MDS) codes, which are optimal in terms of storage efficiency. According to the classification in [3], existing MDS codes for data storage can be categorized...
into two groups. One is called general-purpose codes, which can tolerate any pre-specified number of failures. The other is called special-purpose codes, which can tolerate exactly a certain number of failures, typically two or three.

With regard to special-purpose codes, a noteworthy example is the redundant array of inexpensive disk (RAID) system, which uses multiple disks as a single logical storage unit for fault tolerance. EVENODD [4], RDP codes [5] X-code [6], and Minimal Density RAID-6 codes [7,8] are specially designed for RAID-6, which is required to tolerate double disk failures. Besides, there are also codes specially designed for tolerating triple disk failures [9–14]. A new MDS array code is designed in [15], which can tolerate five disk failures.

Compared to general-purpose codes, they provide a wider range of coding parameters such as code rate and the maximum number of tolerable failures. To describe these codes, we use $n$, $k$, and $L$ to denote the number of encoded packets, the number of source packets and the size of each source packet, respectively.

The most representative class of general-purpose codes is Reed–Solomon (RS) codes introduced in [16], which have encoding and decoding time complexity of $O(n \log^2 n \log \log n)$ [17, Chapter 11.7]. As the codes are defined over a large finite field, encoding and decoding operations are computationally expensive, which prevents it from being widely used in practical data storage systems. Another class of general-purpose MDS codes is proposed in [18], which has a decoding complexity of $O(k^4 L^3)$. However, the decoding complexity is cubic in terms of the packet size $L$, which is usually a large number in data storage systems. In [19], Blaum et al. construct a family of MDS array codes aiming to tolerate multiple disk failures, while they only provide efficient encoding and decoding procedures for the case of two or three disk failures. Its decoding complexity for general cases increases exponentially with the number of erasures. There is another class of standard general-purpose erasure codes, low-density parity-check (LDPC) codes, which achieve attractive decoding performance. Plank and Thomason investigate the application of LDPC codes in data storage systems in [20]. Their experiment results show that LDPC codes achieve good performance only when $k$ is large. Reed–Solomon codes clearly outperform LDPC codes when $k \leq 36$. Digital fountain codes, such as LT codes, also have been studied [21–23]. To apply LT codes to data storage, the challenge is that there is a small probability that a user may not be able to recover the original data from any $k$ survived data packets. Cao et al. proposed an LT-code-based secure cloud storage service (LTCS) in [24]. Exhaustive search is used to obtain a feasible code instance. Though the decoding complexity of LTCS is as low as LT codes, which is nearly $O(k \log k)$, the exhaustive search method is time consuming and sometimes not even be able to output a valid LT code. As it is too complicated to estimate the time of obtaining a valid LT code for a specified storage system, we do not compare the performance of our proposed codes with LT codes in this paper. Cauchy-RS codes are another class of general-purpose MDS codes [25], which improves the RS codes with two modifications. First, the generator matrix becomes the Cauchy matrix instead of the Vandermonde matrix. Second, multiplication in finite field is replaced by the exclusive-or (XOR) operation. The decoding complexity of Cauchy-RS codes is $O(k \omega \log L)$, where $\omega$ is the word size. Its performance can be further improved by optimizing the Cauchy distribution matrix [26]. As Cauchy-RS codes are widely accepted as good general-purpose MDS codes, we use it as the benchmark for evaluating the performance of our proposed codes.

In this paper, we propose a new class of general-purpose erasure codes based on the concept of zigzag decoding [27]. The encoding and decoding process of ZD codes involve only the bit-shift and XOR operations. Decoding is done by zigzag decoding, with time complexity of $O(k^2 L)$ or equivalently $O(kM)$, where $M$ is the total number of information bits. In other words, the time complexity per information bit is $O(k)$. As no finite-field multiplication is needed and the decoding complexity per information bit in terms of XOR is linear, decoding is very fast. We focus only on time complexity. Space complexity of decoding ZD codes is considered in [28].

This paper is an extension of our preliminary work in [29]. It provides a much more comprehensive treatment on ZD codes. Our contributions are summarized as follows:

- We provide a detailed, rigorous proof of an essential condition called increasing-difference property for constructing feasible ZD codes. Two feasible constructions for ZD codes are proposed. One is Vandermonde-matrix based ZD codes and the other is Hankel-matrix based ZD codes.
- We compare the encoding and decoding performance of RDP codes, Cauchy-RS codes and ZD codes by both mathematical analysis and software implementation. As the Jerasure library in [30] implements the fastest version of Cauchy-RS codes, we employ this library to obtain the performance of Cauchy-RS codes. We conduct extensive experiments on encoding and decoding performance evaluation of the three codes. We not only compare their performance under ordinary settings of storage systems, but also investigate the potential and restriction of ZD codes in a wide variety of scenarios. Our experimental results show that ZD codes outperform Cauchy-RS codes in most scenarios.
- We compare the storage overhead of the two different ZD code constructions. Under the ordinary settings of practical storage systems, the storage overhead required of Hankel-matrix based ZD codes is less than 0.7% of the total storage amount of the system and is much smaller than the overhead of Vandermonde-matrix based ZD codes.

The rest of this paper is organized as follows. In section 2, we provide a short introduction of the zigzag decoding concept. In Section 3, we briefly review the ideas of RDP codes and Cauchy-RS codes. In Section 4, we explain the idea of ZD codes and the two construction methods. In Section 5, we present the details of implementation and the parameter settings. In Section 6, we analyze the computational complexities of the encoding and decoding algorithms of the three codes. In Section 7, we provide numerical results and performance evaluation of different coding schemes. Section 8 concludes our paper.
2. Related work on zigzag decoding

The zigzag decoding concept is first introduced in [27] to combat the hidden terminal problem in wireless networks. As shown in Fig. 1, x and y are source packets sent by two senders who are unable to sense each other. Let $p_1$ and $p_2$ be two received packets obtained by the superpositions of two “collided” packets with different offsets. The first bit $x_1$ is interference-free in $p_1$, so it can be directly read out. After subtracting $x_1$ from $p_2$, $y_1$ can be obtained. Repeat this process until every bit of x and y is decoded. The access to the two collision packets follows a zigzag pattern. In this scenario, the maximum number of overlapped source packets to perform zigzag decoding is two. Similar to the idea of zigzag decoding, a triangular network coding scheme was proposed for the wireless broadcasting problem in [31], with an aim of obtaining more innovative encoding packets. The concept of shift operations are also employed to design packet-level forward error correcting codes in [32], but their decoding process is based on matrix inversion, which is different from the zigzag decoding algorithm in this paper. Besides, the same idea of introducing extra overhead to parity packets can also be found in the construction of a regenerating code in [33].

The idea of zigzag decoding can be applied to the design of general-purpose erasure codes. Motivated by the low time complexity of zigzag decoding, we design the first class of zigzag decodable (ZD) codes in [29]. Our designed codes have the same erasure fault tolerance capability as MDS codes, but strictly speaking, they are not MDS codes due to the extra overhead incurred by the bit-shift operation to be described in Section 4. To avoid misunderstanding, we define the notion of $k$-reliability, which can be used to describe the erasure fault tolerance of both ZD codes and MDS codes:

**Definition 1.** An $(n, k)$ code is said to be $k$-reliable if after encoding $k$ information symbols into $n$ coded symbols, the $k$ information symbols can be recovered by any $k$ of the coded symbols.

ZD codes have been applied to various situations. For example, they can be used as a network coding scheme for the combination network [34]. Besides, a fountain code based on ZD codes is proposed in [35] and is shown to outperform Raptor codes in terms of the transmissions overhead. Since ZD codes are general-purpose erasure codes, in principle, they can be used in any scenario which uses existing erasure codes as one of the building components. An example is the STAIR code, which is recently proposed for tolerating device and sector failures in practical storage systems [36].

Apart from pursuing excellent encoding and decoding performance for data retrieval problems, how to repair failed nodes is also an important issue in maintaining the reliability of data storage systems. Repairing schemes for single-failure cases of RDP and EVENODD codes are proposed in [37,38]. Zhu et al. proposed an efficient repair algorithm for single-node failure for general MDS array codes in [39]. Hou et al. proposed the BASIC code for both functional repair and exact repair [40]. In this paper, we focus on the encoding and decoding speeds of the data storage and retrieval processes. We remark that ZD codes can be used as the outer code of the Fractional Repetition codes, which support exact and uncoded repair [41,42]. Designing ZD codes with efficient repair property is an interesting topic for future work.

3. MDS codes for data storage systems

The MDS codes are known to be optimal in terms of reliability-redundancy tradeoff. A comprehensive performance comparison among different MDS codes is made in [3]. Based on their evaluation, the encoding and decoding performance of RDP outperforms other codes when the number of parity symbols is two. As the other special-purpose erasure codes such as STAR codes [43] and generalized EVENODD codes are very similar to RDP codes, while our proposed codes are designed for tolerating multiple failures, we treat RDP as a representative of special-purpose codes and use it as a benchmark for comparison.

For general-purpose codes, as discussed in the previous section, Cauchy-RS codes are the most efficient general-purpose erasure codes. Therefore, we studied in detail the performance comparison between ZD codes and Cauchy-RS codes. The performance of Cauchy-RS is sensitive to the size of the finite field. The larger the number of source packets or parity packets, the larger the required field size and the worse it performs.

Since we choose RDP and Cauchy-RS codes as the counterpart for the performance analysis of ZD codes, in this section we briefly review the general ideas of these two classical coding techniques.
3.1. Cauchy-RS codes

Cauchy-RS codes improve the original RS codes with two modifications. Instead of using Vandermonde matrices, Cauchy-RS Codes employ the Cauchy matrices as the generator matrix for the parity packets. Encoding $k$ packets into $m$ encoded parity packets, the $m \times k$ Cauchy matrix over $\text{GF}(2^\omega)$ is stated as

\[
\begin{bmatrix}
\frac{1}{x_1+y_1} & \frac{1}{y_1+y_2} & \cdots & \frac{1}{x_1+y_k} \\
\frac{1}{x_2+y_1} & \frac{1}{y_1+y_2} & \cdots & \frac{1}{x_2+y_k} \\
\vdots & \vdots & \ddots & \vdots \\
\frac{1}{x_m+y_1} & \frac{1}{y_1+y_2} & \cdots & \frac{1}{x_m+y_k}
\end{bmatrix}
\]

where $\{x_1, x_2, \ldots, x_m\}, \{y_1, y_2, \ldots, y_k\}$ are two subsets of elements in $\text{GF}(2^\omega)$, where $\omega$ is called the word size. Note that the term “word” refers to the number of bits that are grouped together as a single unit for encoding and decoding. For any $i \in \{1, 2, \ldots, m\}$ and any $j \in \{1, 2, \ldots, k\}$, the elements satisfy $x_i \neq y_j$. By the pigeon hole’s principle, the field size has to satisfy $2^\omega \geq m + k = n$ [3]. Every square sub-matrix of a Cauchy matrix is nonsingular. This property ensures that the original message can be decoded from any $k$ packets. The expensive multiplication in large Galois Field is converted to binary operation by representing each element in $\text{GF}(2^\omega)$ as a $\omega \times \omega$ binary matrix. Therefore, an $m \times k$ generating matrix over $\text{GF}(2^\omega)$ is converted to an $\omega m \times \omega k$ matrix over $\text{GF}(2)$. More details can be found in [3,25,26].

3.2. RDP codes

Fig. 2 shows an illustrative example for RDP codes. There are four data disks, $D_0, D_1, D_2$ and $D_3$, and each data disk consists of four data bits. Take $D_0$ for instance, the four data bits are $d_{0,0}, d_{1,0}, d_{2,0}$ and $d_{3,0}$, which constitute a word. There are two parity disks. Each parity disk contains the same number of bits as a data disk. The first one is the row parity disk, denoted as $P$. Each parity bit is just the XOR of all the data bits in the corresponding row (see the upper part of Fig. 2). The other one is the diagonal parity disk, denoted as $Q$. The calculation of its parity bits involves not only the data packets, but also the bits in the row parity disk, $P$. As shown in the lower part of Fig. 2, the bits in the four data disks and the row parity disk, $P$, are classified into five diagonal sets. Each bit in the diagonal parity disk is the XOR of all the other elements in the same diagonal set. Take the parity bits $q_0$ as an example, it is the XOR of all the elements in the first diagonal set, which contains $d_{0,0}, d_{3,2}, d_{2,3}$ and $p_1$. In this example, the word size, $\omega$, is equal to 4, since four data bits are grouped together to form a single unit for encoding and decoding, as can be seen from Fig. 2 that a word of four bits is shown for each disk. For RDP codes, the word size, $\omega$, has to satisfy two conditions: (i) $\omega \geq k$ and (ii) $\omega + 1$ is a prime number. RDP codes have excellent performance when $\omega = k$, but the performance degrades as $(\omega - k)$ increases [3].

![Fig. 2. Example of RDP codes when $k = 4$ and $\omega = 4$.](image-url)
4. Zigzag decodable codes

ZD Codes are designed as a class of general-purpose codes. The encoding and decoding process of ZD codes involve only bit-shift and XOR operations. They have two desirable properties, namely, k-reliability and zigzag decodability. Zigzag decodability means that the original file can be recovered by zigzag decoding. Before formally defining zigzag decoding, we first describe a mathematical representation of ZD codes. Given a source file $s$, we split it into $k$ packets $s_1, s_2, \ldots, s_k$. Each packet $s_i$ consists of $L$ bits, and is represented by the following polynomial,

$$s_i(z) = s_{i,0} + s_{i,1}z + s_{i,2}z^2 + \cdots + s_{i,L-1}z^{L-1},$$

where $s_{i,j}$ is an element of $\mathbb{GF}(2)$. The $k$ source packets are linearly combined to obtain $n$ coded packets $c_1(z), c_2(z), \ldots, c_n(z)$, each of which has $L + l$ bits, where $l$ denotes the maximum offset. The coded packet $c_i(z)$ is given by

$$c_i(z) = z^{a_{1,i}}s_1(z) + z^{a_{2,i}}s_2(z) + \cdots + z^{a_{k,i}}s_k(z),$$

where $i \in [1, n]$, and $t_{i,j} \in \mathbb{Z}$. Note that $l = \max_{i,j}a_{i,j}$. In theory, the coded packets can be of different lengths. For ease of analysis and presentation, we simply assume that dummy bits are appended to coded packets, so that they are of equal length. If systematic codes are used, we assume that the source packets are of length $L$ while parity packets are of length $L + l$. For practical implementation, it is also more convenient to have equal-length parity packets.

The encoding procedure in (3) can be represented by the following matrix form,

$$c(z) = A(z)s(z),$$

where $c(z)$ is an $n$-dimensional vector whose $i$-th element is $c_i(z)$, $s(z)$ is an $k$-dimensional vector whose $j$-th element is $s_j(z)$, and $A(z)$ is an $n \times k$ matrix whose $(i,j)$-th element is $z^{a_{i,j}}$. Our aim is to design a systematic erasure code. Define $m = n - k$, which represents the number of parity packets. The encoding matrix $A(z)$ should be in the form of

$$A(z) = \begin{bmatrix} I_{k\times k} \\ B(z) \end{bmatrix},$$

where $I_{k\times k}$ is the $k \times k$ identity matrix and $B(z)$ is an $m \times k$ matrix whose $(i,j)$-th element is $\alpha_{i,j}(z) = z^{a_{i,j}}$.

Now it remains to design the matrix $B(z)$. First of all, the code should be $k$-reliable, which implies that packets encoded by any square submatrix $M(z)$ of $B(z)$ can be used to recover the source packets. Suppose the size of the given square matrix $M(z) = [z^{l_{i,j}}]$ is $k \times k$, and we define $\mathcal{M} = [1, 2, \ldots, k]$. Besides, we introduce a $k \times k$ integer matrix $T = [t_{i,j}]$, whose element is the exponent of the corresponding element in $M(z)$. If the recovering process can be done by Algorithm 1, we say that the code is zigzag decodable. Note that the input to Algorithm 1 are $k$ coded parity packets and the matrix $T$, while the output are the $k$ source packets. The basic idea is to search for a coded packet that has an “exposed” bit, which can be directly read out. After that, the bit is subtracted from other coded packets. The procedure repeats until all source bits are decoded. An example to illustrate the execution of Algorithm 1 can be found in the Appendix.

Remark 1. In Algorithm 1, steps 18 and 19 are needed when all the $L$ bits of source packet $i$ have been decoded. In that case, the corresponding element, $B[i][j^*]$, is updated to a value that is large enough so that it will never be chosen in Step 12.

Remark 2. The computational complexity of Algorithm 1 is $O(k^2L)$ due to the two for-loops in lines 10 and 15, respectively.

Remark 3. For practical implementation, the algorithm can be further sped up, as the sequence of coded packets that have “exposed” bits is eventually periodic (see the example in the Appendix). This feature has been incorporated into our software used in Section 5.

Now we are going to prove a sufficient condition on $M(z)$ for zigzag decodability. For $i \in \mathcal{M}$, define

$$S_i = \arg\min_{j \in \mathcal{M}} t_{i,j}.$$

Note that $S_i$ is a set of integers. We say that $S_u \times S_v$ if the smallest element in $S_u$ is greater than or equal to the largest element in $S_v$.

Definition 2 (Increasing difference property). A square matrix $M(z) = [z^{l_{i,j}}]$ and its corresponding exponent $T = [t_{i,j}]$ are both said to satisfy the increasing difference property if given any $i < i'$ and $j < j'$, we have

$$t_{i,j'} - t_{i,j} < t_{i',j'} - t_{i',j}.$$
Algorithm 1 ZigZag decoding algorithm.

Require: \( k \) binary arrays, \( Y_1, Y_2, \ldots, Y_k \), each of length \( L+i \), and a \( k \times k \) integer array, \( T \)
Ensure: \( k \) binary arrays, \( X_1, X_2, \ldots, X_k \), each of length \( L 

// Initialization
1: \( B := T \);
2: Let \( V \) be the array of size \( L+i \) whose first \( L \) elements are equal to 1 and the last \( i \) elements are equal to 0;
3: for \( i = 1 \) to \( k \) do
4: \( p_i := 1 \);
5: for \( j = 1 \) to \( k \) do
6: \( V_j \) be obtained by cyclically shifting \( V \) to the right by \( T[i][j] \) positions;
7: end for
8: \( A_j := V_1 + V_2 + \cdots + V_k \);
9: end for
10: for number of decoded bits = 1 to \( l \) do
11: Find the smallest \( i' \) such that \( A_i[p_i] = 1 \);
12: \( j' := \arg\min_{j} B[i'][j] \);
13: \( b := Y_j[p_i] \) and \( h := p_i - T[i'][j'] \);
14: \( X_j[h] := b; \)
15: for \( i = 1 \) to \( k \) do
16: \( p_i := b + T[i'][j'] \) and \( Y_j[p_i] := Y_j[p_i] \oplus b; \)
17: \( B[i][j'] := B[i][j'] + 1; \)
18: if \( B[i][j'] - T[i][j'] - L = 0 \) then
19: \( B[i][j'] := L + l + 1; \)
20: end if
21: if \( A_i[p_i] > 1 \) then
22: \( A_i[p_i] := A_i[p_i] - 1; \)
23: else
24: \( p_i := p_i + 1; \)
25: end if
26: end for
27: end for

This condition means that the relative offset between source packet \( j' \) and \( j \) are increasing from parity packet \( i \) to \( i' \). Graphically, the source packets in parity packet \( i \) are “sliding” to the right to obtain parity packet \( i' \), as can be seen from the example in the Appendix (c4, c5, c6 and c7 in Fig. 13). This condition ensures that there is always a parity packet that has an “exposed” bit so that zigzag decoding can proceed. A proof now follows.

Lemma 1. Given any \( k \times k \) matrix \( T \) that has the increasing difference property, \( S_1, S_2, \ldots, S_k \) as defined in (6) satisfy the following properties:

(i) \( S_i \supset S_i \) if \( i \leq i' \).

(ii) \( |S_i \cap S_{i+1}| \leq 1 \) for \( i = 1, 2, \ldots, k - 1 \).

(iii) \( \bigcup_{i \in \mathcal{A}} S_i \) can be partitioned into the following \( k \) disjoint subsets: \( S_1 \setminus S_2, S_2 \setminus S_3, \ldots, S_{k-1} \setminus S_k, S_k \).

Proof. (i) The first property is equivalent to this: If \( j \in S_i \) and \( j' \in S_i' \), where \( i < i' \), then \( j \geq j' \). We prove by contradiction by first assuming \( j < j' \). Then we have

\[ t_{i, j} - t_{i, j} < t_{i', j'} - t_{i', j} \leq 0, \]

where the first inequality follows from the increasing difference property and the second one follows from the fact that \( j' \in S_i \). As a consequence, we have

\[ t_{i, j} > t_{i, j'}, \]

which contradicts with the fact that \( j \in S_i \).

(ii) clearly follows from (i).

(iii) We prove by mathematical induction. It is clear that \( S_1 \cup S_2 \) can be partitioned into \( S_1 \setminus S_2 \) and \( S_2 \). Assume that \( \bigcup_{i=1}^{u} S_i \) can be partitioned into

\[ S_1 \setminus S_2, S_2 \setminus S_3, \ldots, S_{u-1} \setminus S_u, S_u. \]

for \( u \geq 2 \). Consider \( \bigcup_{i=1}^{u+1} S_i \). It can be partitioned into

\[ S_1 \setminus (S_2 \cup S_{u+1}), S_2 \setminus (S_3 \cup S_{u+1}), \ldots, S_{u-1} \setminus (S_u \cup S_{u+1}), S_u + 1. \]

Since (i) and (ii) together imply that for \( i = 1, 2, \ldots, u - 1 \),

\[ S_i \setminus (S_{i+1} \cup S_u) = S_i \setminus S_{i+1}, \]

the statement is proved. □
**Theorem 2.** Given $k$ coded packets $Y_1, Y_2, \ldots, Y_k$ and the encoding matrix $M(z)$, which has the increasing difference property, zigzag decoding as stated in Algorithm 1 is able to recover source packets $X_1, X_2, \ldots, X_k$.

**Proof.** Zigzag decoding can successfully recover the source packets if it can always find an interference-free symbol of a source packet from one of the encoded packet, or more formally, if it can always find $i^*$ in Step 11 of Algorithm 1 during each iteration. We prove by contradiction.

Recall that in Algorithm 1, $B$ is initialized to $T$, which has the increasing difference property. After every iteration, $B$ will be updated by adding an all-one vector to one of its columns. It is clear that such an operation keeps the increasing difference property unchanged. In other words, $B$ always satisfies the increasing difference property.

Suppose the algorithm fails in Step 11 during a certain iteration. Based on the current value of $B$, we define $S_i$ for all $i \in \mathcal{M}$ as in (6). Since the algorithm halts abnormally, we must have $|S_i| \geq 2$ for all $i$, for otherwise $|S_i| = 1$ for some $i$ and an “exposed” bit can be found. By Lemma 1(iii), we have

$$\left| \bigcup_{i \in \mathcal{M}} S_i \right| = \sum_{i=1}^{k-1} |S_i \setminus S_{i+1}| + |S_k|.$$ 

Since $|S_i| \geq 2$ for all $i$, by Lemma 1(ii), we have

$$\left| \bigcup_{i \in \mathcal{M}} S_i \right| \geq (k - 1) + 2 = k + 1 > k,$$

which is a contradiction. Therefore, the algorithm can always find $i^*$ in Step 11, which implies that we can always find an exposed bit and decode it out. The algorithm stops only when all bits of all source packets are determined. □

The following two constructions of ZD codes are designed based on this essential condition. The first one is inspired by Vandermonde matrix. We name the it as Vandermonde-matrix based ZD codes. The second one is derived from an Hankel matrix, so we call it Hankel-matrix based ZD codes.

### 4.1. Vandermonde-matrix based ZD codes

We now demonstrate a construction of ZD codes based on Vandermonde matrix. As stated before, our goal is to design an encoding matrix as in (5). In other words, we need to design the structure of $B(z)$. Since the core idea of designing $B(z)$ is to assign the offset of each source packet, which is represented by the exponent of each element in $B(z)$. For convenience, we introduce an integer matrix $P$, whose element is the exponent of the corresponding element in $B(z)$. Therefore, designing a code construction $B(z)$ is equivalent to designing $P$.

In our construction, the $m \times k$ matrix $B(z)$ takes the following form:

$$
\begin{bmatrix}
1 & 1 & \cdots & 1 \\
1 & z & \cdots & z^{k-1} \\
1 & z^2 & \cdots & z^{m-1} \\
1 & z^3 & \cdots & z^{(m-1)(k-1)}
\end{bmatrix},
$$

(7)

and the corresponding integer matrix $P$ is given by

$$
\begin{bmatrix}
0 & 0 & \cdots & 0 \\
0 & 1 & \cdots & k - 1 \\
0 & 2 & \cdots & \\
0 & m - 1 & \cdots & (m - 1)(k - 1)
\end{bmatrix}.
$$

(8)

It is easy to check that any square submatrix of $B(z)$ satisfies the increasing difference property. As a result, the code is zigzag decodable.

Note that the attractive low decoding complexity of ZD codes needs to sacrifice slight storage efficiency. For this code, the longest parity packet contains $\ell_P = (m - 1)(k - 1)$ more bits than the source packets. There are totally $m$ parity packets, the proportion of extra storage overhead is at most $\frac{m(m - 1)(k - 1)}{(k + m)L}$.

### 4.2. Hankel-matrix based ZD codes

Next, we consider another construction of ZD codes based on Hankel matrix, which has a smaller storage overhead. We will define the code structure by means of the integer matrix $P$, which is derived from the class of Hankel matrix.
Definition 3. An $N \times N$ matrix $H$ is called a Hankel matrix if it is in the following form,

$$
\begin{bmatrix}
  h_0 & h_1 & h_2 & \cdots & h_{N-1} \\
  h_1 & h_2 & h_3 & \cdots & h_N \\
  h_2 & h_3 & h_4 & \cdots & h_{N+1} \\
  \vdots & \vdots & \vdots & \ddots & \vdots \\
  h_{N-1} & h_N & h_{N+1} & \cdots & h_{2(N-1)}
\end{bmatrix}
$$

(9)

For $i, j \in \{1, 2, \ldots, N\}$, the $(i, j)$-th element of $H$, $h_{i,j}$, satisfies

$$
h_{i,j} = h_{i+j-2}
$$

(10)

for some given sequence $h_0, h_1, \ldots, h_{2(N-1)}$.

Given the value of $N$, in our code design, the values of $h_0, h_1, \ldots, h_{2(N-1)}$ are specified as follows:

$$
h_{N-1} = 0,
$$

(11)

and for $i = 0, 1, 2, \ldots, 2N - 3$,

$$
h_{i+1} - h_i = i - N + 2.
$$

(12)

For example, when $N = 4$, we have $h_0 = 3, h_1 = 1, h_2 = 0, h_3 = 0, h_4 = 1, h_5 = 3$ and $h_6 = 6$. According to (11) and (12), the Hankel matrix $H_{4 \times 4}$ is given by

$$
\begin{bmatrix}
  3 & 1 & 0 & 0 \\
  1 & 0 & 0 & 1 \\
  0 & 0 & 1 & 3 \\
  0 & 1 & 3 & 6
\end{bmatrix}
$$

(13)

Note that according to (11), the anti-diagonal of the matrix is always equal to zero. By (12) with $i = N - 2$, the anti-superdiagonal is also equal to zero.

Our code is constructed by specifying the $m \times k$ matrix $P$, which is obtained from a submatrix of the $N \times N$ Hankel matrix constructed by (11) and (12), where $N = \max(m, k)$. If $m < k$, $P$ is the submatrix consisted of the $m$ rows of $H$ starting from the row of index $\lfloor (k - m)/2 \rfloor + 1$. Otherwise, $P$ is the submatrix consisted of the $k$ columns of $H$ starting from the column of index $\lfloor (m - k)/2 \rfloor + 1$. For example, if $k = 3$ and $m = 4$, then $P$ should be the submatrix consisted of three columns starting from the column with index $\lfloor (m - k)/2 \rfloor + 1 = 1$. According to (13), $P$ is given by

$$
\begin{bmatrix}
  3 & 1 & 0 \\
  1 & 0 & 0 \\
  0 & 0 & 1 \\
  0 & 1 & 3
\end{bmatrix}
$$

(14)

This is actually the code we considered in Example 1.

Now we are going to prove that the matrix $H$ constructed satisfies the increasing difference property.

Theorem 3. Given a Hankel matrix constructed by (11) and (12), if $i < i'$ and $j < j'$, then

$$
h_{i,j} - h_{i,j} < h_{i',j'} - h_{i',j'}
$$

(15)

Proof. Consider first the case when $j' - j = 1$. By (10) and (12), we have

$$
h_{i,j} - h_{i,j} = h_{i+j-1} - h_{i+j-2}
\begin{align*}
&= (i + j - 2) - N + 2 \\
&< (i' + j - 2) - N + 2 \\
&= h_{i'+j-1} - h_{i'+j-2} \\
&= h_{i,j'} - h_{i',j'}.
\end{align*}
$$

Now consider the general case when $j' > j$. The above result implies the following set of inequalities:
Fig. 3. Illustration of the Hankel matrix for the case when $m \geq k$.

\[
\begin{align*}
  h_{i,j+1} - h_{i,j} &< h_{i',j+1} - h_{i',j} \\
  h_{i,j+2} - h_{i,j+1} &< h_{i',j+2} - h_{i',j+1} \\
  &\vdots \\
  h_{i,j'} - h_{i,j'-1} &< h_{i',j'} - h_{i',j'-1}
\end{align*}
\]

Summing them up, we obtain (15). \qed

For Hankel-matrix based ZD codes, the maximum offset $l_H$ is the value of the bottom right element of matrix $P$, which is equal to $h_j$ for some $j$ between $N - 1$ and $2(N - 1)$. Due to the symmetry of $H$, it suffices to consider only the case when $m \geq k$. Consider Fig. 3. Since the bottom left element of $H$ is $h_{N-1}$, the bottom left element of $P$ is $h_i$, where $i = (N - 1) + \lfloor (m - k)/2 \rfloor$ and the bottom right element of $P$ is $h_j$, where

\[
j = i + k - 1
\]

\[
= (N - 1) + \left\lfloor \frac{m + k}{2} \right\rfloor - 1
\]

\[
= (N - 1) + \left\lfloor \frac{n}{2} \right\rfloor - 1.
\]

According to (11) and (12), the maximum offset is given by

\[
l_H = h_j = \sum_{i=N-1}^{j-1} (i - N + 2)
\]

\[
= \sum_{k=1}^{\lfloor n/2 \rfloor - 1} k
\]

\[
= \frac{1}{2} \left( \left\lfloor \frac{n}{2} \right\rfloor - 1 \right) \left( \frac{n}{2} \right).
\]

Consider the previous example when $k = 3$ and $m = 4$. It can be directly observed from (14) that the maximum offset, $l_H$, is equal to 3, which agrees with the above expression.

Recall that the maximum offset of Vandermonde-matrix based ZD codes is

\[
l_V = (m - 1)(k - 1).
\]

Depending on the values of the code parameters, it can be larger or smaller than $l_H$. For example, if $m$ is a fixed value, while $n$ and $k$ grow, then $l_V$ grows linearly with $k$ while $l_H$ grows quadratically with $n$. Vandermonde-matrix based ZD codes clearly have a smaller storage overhead.

On the other hand, suppose we keep the proportion of parity packet keeps constant, i.e., $k = \alpha m$ for some fixed positive rational number $\alpha$. In other words, the code rate $R$ is a constant equal to $\alpha/(\alpha + 1)$. Then, we have

\[
l_H \leq \frac{1}{2} \left( \frac{n}{2} - 1 \right) \frac{n}{2}
\]

\[
= \frac{n(n - 2)}{8}
\]

\[
= \frac{(1 + \alpha)^2 m^2}{8} - \frac{(1 + \alpha)m}{4}
\]
The maximum offset of the Vandermonde-matrix based ZD codes is

\[ l_V = \alpha m^2 - (1 + \alpha)m + 1. \]

The difference of the maximum offsets of the two codes is

\[ l_V - l_H = -\frac{1}{8}m^2[(\alpha - 3)^2 - 8] + O(m). \]  \hspace{1cm} (16)

Hankel-matrix based ZD codes perform better if \( \alpha \) satisfies \( 3 - 2\sqrt{2} \leq \alpha \leq 3 + 2\sqrt{2} \), or \( 0.17 \leq \alpha \leq 5.83 \). This corresponds to the case when the code rate satisfies 0.15 \( \leq R \leq 0.85 \). For practical applications, \( \alpha \), or equivalently, \( R \), typically lies within the above range, so Hankel-matrix based ZD codes should have a lower storage overhead. Numerical comparisons of the extra storage overhead of these two codes for some common parameters for storage systems will be presented in Section 7.

5. Experiment settings

In this section, we describe the experiment settings, including the process of encoding and decoding, and settings of involved parameters. The input of the encoding experiment is a large source file with size around 1 GB. The output is a set of files with smaller size. To avoid confusion, we call these small files as subfiles, including \( k \) source subfiles and \( m \) parity subfiles. The \( k \) source subfiles are obtained by splitting the large source file into \( k \) pieces. The parity subfiles are calculated by different coding schemes. The input of the decoding experiment is a set of \( k \) subfiles selected from the \((k + m)\) subfiles, i.e., the number of lost subfiles is set as \( m \). Although the number of erasures can be less than or equal to \( m \), and the erasures can happen to not only source subfiles, but also parity subfiles, losing \( m \) source subfiles is the most complex case in the decoding process of every coding scheme. And recovering the source file in high speed with most tolerable erasures is an important property in erasure code design. Thus we test the decoding speed of all coding schemes based on the case when \( m \) source subfiles are lost. Based on modern erasure-coded systems, such as Google File System [44,45], Facebook Hadoop Distributed File System [46] and Windows Azure Storage [47], and the analysis and discussion in [48–50], commonly used erasure codes have storage amounts of 1.33× to 2× of the original data file. Note that the storage amount of the traditional three-replication code is 3× of the data file size and the purpose of using other erasure codes is to reduce this 3× storage amount. We choose the combinations of \((k, m)\) that are likely used in actual storage systems for our experiments. (6, 2) is the combination appeared in RAID-6 system. (6, 3) is chosen by Google II [44,45]. (10, 4) is the preference for HDFS-RAID in Facebook [46]. (12, 4) is considered to achieve good performance in Windows Azure Storage [47]. We also investigate the trend of encoding and decoding speeds as increasing number of source subfiles with four fixed ratios of storage overhead, which are 1.33×, 1.6×, 1.8×, and 2.0×. For every coding technique, each of these ratios has two corresponding curves describing its encoding and decoding speeds respectively. We choose four pairs of \((k, m)\) for each curve. All the values of \((k, m)\) taken for our experiments are stated in Table 2.

In many applications, the source file to be stored is often much larger than the memory size of most computers. In our experiments, we set the source file size as 1 GB. To encode such a large file, using a buffer to complete the process in a batch-wise fashion is a common solution. First, the data of source file are written into the buffer sequentially, until the buffer is full. The data in the buffer are then encoded. After encoding, the source data and parity data are written to the corresponding subfiles. The process is repeated until the whole file is encoded. Based on the results in [3], we set the buffer size to be the ballpark of 600 KB, which is sufficiently large to achieve efficient I/O. The parameters of source file size and buffer size are summarized in Table 2.

Now we illustrate the encoding of the data in the buffer. The buffer is partitioned into several contiguous memory space of equal size. Each of them is called a stripe. As shown in Fig. 4, the buffer consists of \( s \) stripes. In each stripe, there are \( k \) data blocks, \( D_0, D_1, \ldots, D_{k-1} \) and \( m \) parity blocks, \( P_0, P_1, \ldots, P_{m-1} \). The data in the \( m \) parity blocks are calculated from the data temporarily stored in the \( k \) source blocks based on a specific coding method. Apparently a stripe is a minimum collection of bits that encode and decode together. After encoding has finished, the data in the buffer will be written into the corresponding subfiles.

The reason for introducing the concept of stripe is that performing XOR operations over a group of bits is much faster than over a single bit. Also, load balancing can be easily realized with stripes. The size of one stripe depends on the size of each block. If we denote the size of source blocks as \( L \) bytes and the size of parity blocks as \( L' \) bytes, then the stripe has a size of \((kL + mL')\) bytes. For ordinary coding schemes, \( L \) equals \( L' \), while it is not the case in our ZD codes. Since in ZD codes, we allow a little overhead in parity blocks, we have \( L' > L \). Once the block size is set, the number of stripes is only determined by the buffer size. In our experiment, \( L \) is set to be 4 KB and \( L' \) is calculated according to the code construction of ZD codes. The chosen data block size is also stated in Table 2. In our implementation, the data in one stripe is allocated in consecutive address space to obtain good data locality. Data locality is the space difference of a sequence of memory access performed by a processor. Once a memory location is accessed, the processor will load a chunk of data in memory around the accessed location into the cache. Therefore, good data locality reduces cache miss and results in better performance.

The remaining problem is how to determine the parameter \( \omega \) for RDP codes and Cauchy-RS codes. According to [3], we partition each data block into \( \omega \) segments for Cauchy-RS codes. Thus the data block size equals to \( \omega \) times the size of segment, \( P \). Fig. 5 provides an example for encoding the first parity segment. According to [30], \( \omega \) must be larger than
Note that the performance of Cauchy-RS degrades when \( \omega \) increases, since with larger \( \omega \), the finite field size will be larger and the binary encoding matrix will be more complicated. For this reason, we always choose the smallest feasible \( \omega \) for Cauchy-RS because we want to compare ZD codes with its best case. For fair comparison, we need to set the size of data block in ZD codes the same as it is set in Cauchy-RS codes, so that they perform the same number of read-encoding-write procedure to output final subfiles. Thus the impact on performance caused by reading from and writing into the buffer should be almost the same. Concerning RDP codes, the performance is influenced by the difference between \( \omega \) and \( k \). We also introduce the concept of segment to get the best performance of RDP by setting \( \omega = k \). We still have to ensure the setting of data block size in RDP codes is the same as that in other coding schemes. The decoding process is not going to be discussed in details, since it is just the inverse operation of encoding. The coded subfiles used for decoding is first read into the buffer. After decoding is performed, it will be written to the memory and waits for constructing the original source file.

6. Computational complexity

Now we analyze the computational complexity of encoding and decoding under our experiment settings. Since the time of performing XOR operation over a single bit is the same as performing XOR operation over a computer word, we use ‘XOR’ to denote one XOR operation over a computer word. And the size of an object is measured by computer words. Let the size of the source file be \( M \), the size of the buffer be \( B \), and the size of each data block be \( L \). To encode the whole file, the
same encoding or decoding process in one stripe has to repeat $Mk/B = M/(k \ell L)$ times. We do not count the shift operation, since the encoding and decoding algorithms only need to address the relevant word for XOR operation. Take the encoding of (2, 2) Vandermonde-matrix based ZD code as an example. To generate the $i$-th word of the second parity block for $i \geq 1$, we can simply perform XOR between the $i$-th word of the first source block with the $(i + 1)$-th word of the second source block. It is not necessary to actually perform the shift operation.

For the encoding process of RDP codes, each row parity word requires at most $k - 1$ XORs. There are $\omega$ rows in our array, $\omega(k - 1)$ XORs are required for row parity. Each diagonal parity word contains $k$ data words or row parity words. Thus at most $k - 1$ XORs needed. There are $\omega$ diagonal parity words in an array, so $\omega(k - 1)$ XORs are required for diagonal parity. The number of such arrays in each stripe is equal to the size of segment. The number of XORs performed in one stripe is $2P(\omega(k - 1))$. Therefore, encoding a file of size $M$ using RDP codes requires $2M(k - 1)/k$ XORs in total, which achieves the minimum number of XORs among all MDS array codes [5]. For decoding complexity, reconstructing any two data blocks is actually $2\omega$ words. Each parity set is of size $k$ words, so the cost to reconstruct each word is at most $k - 1$ XORs. The complexity for reconstructing the data blocks in one stripe is $2P(\omega(k - 1))$. With respect to the whole file, the complexity is again $2M(k - 1)/k$ XORs.

According to [25], the encoding complexity of Cauchy-RS for each parity block is $O(kP\omega^2)$ XORs. Therefore, $O(\omega M\ell)$ XORs are required for encoding. Cauchy-RS requires $O(k \omega M \omega^2)$ XORs and $O(m^2)$ multiplications in $GF(2^\omega)$ in each stripe for decoding process. Thus reconstruction of the original file requires $O(\omega M\ell)$ XORs and $O(m^2)$ multiplications in $GF(2^\omega)$.

For ZD codes, $(k - 1)mL$ XORs are required for encoding in each stripe. Since there are $M/(kL)$ stripes, the whole encoding process needs $mM(k - 1)/k$ XORs. If $k > m$, during decoding, $O((k - m)mL)$ XORs are required to subtract the systematic part from the parity blocks in each stripe. Since $(k - m)m \leq 2^mL$, the subtraction step requires $O(k^2L)$. Then zigzag decoding is used to decode the $m$ parity blocks, which requires $O(m^2L)$ XORs (see Remark 2 of Algorithm 1). The operation in one stripe should be repeated $M/(kL)$ times to finish decoding. The whole decoding process involves $O(kM + \frac{mM}{k})$ XORs. If $k < m$, all systematic subfiles are lost. Performing the Zigzag decoding algorithm requires $O(k^2L)$ XORs (again see Remark 2 of Algorithm 1). Thus we need $O(kM)$ XORs for overall decoding. In conclusion, the total XORs required for overall decoding is at most $O(kM + \frac{mM}{k})$. Note that for $m$ proportional to $k$, both encoding and decoding complexities are linear. For the case of double failure, i.e., $m = 2$, the encoding complexity of ZD codes becomes $O(M)$, which is the same as that of RDP codes. This agrees with our numerical results to be presented in the next section.

We summarize the encoding and decoding complexities of the three codes in Table 1. Recall that RDP applies only to the case when $m = 2$, so we first compare the performance of the three codes under this condition. For Cauchy-RS, the word size, $\omega$, has to satisfy $2^\omega \geq n$. When $m = 2$ and $k$ can be arbitrarily large, RDP and ZD have the same order of encoding complexity, which is lower than that of Cauchy-RS. Besides, RDP has the lowest decoding complexity. In other words, RDP has the best performance, which is expected as it is a special-purpose codes, which can only be used when $m = 2$.

Next, we consider the general case. We compare the performance between Cauchy-RS and ZD. It can be seen that ZD has lower encoding complexity than Cauchy-RS. For decoding, Cauchy-RS requires XOR and multiplication over $GF(2^\omega)$, while ZD requires XOR only. The time complexity of multiplication depends on the implementation. For multiplication over finite field of characteristic 2, the most straightforward way is to simply perform bit-shift and bit-wise addition, which has time complexity of $O(\omega^2)$. If the Karatsuba–Ofman Algorithm [51,52] is used, then it can be improved to $O(\omega\log_2^3)$. As a result, the multiplication can be done in $O(m^2\omega\log_2^3)$ XORs. If $M$ is fixed, then the decoding complexity of ZD codes is lower than that of Cauchy-RS. On the other hand, if $M$ is large, then the time consumption of multiplication can be ignored. In this case, if $m$ is fixed while $k$ increases, then Cauchy-RS requires fewer XORs than ZD. If the relative storage overhead, $m/k$, is fixed and we let both $m$ and $k$ goes to infinity, then the decoding complexity of ZD is linear and is lower than that of Cauchy-RS by a factor of $\omega = \log n$.

### 7. Experiment results

While the computational complexity is obtained in the previous section, it is still necessary to obtain precise numerical results to evaluate their encoding and decoding speeds. Therefore, we implement ZD codes and RDP codes in C to test their encoding and decoding performance. For Cauchy-RS codes, Jeasure library in [30] is regarded as the best implementation. So we just use this library to realize Cauchy-RS codes in our experiment. According to the description in Section 4, we summarize the parameter settings in Table 2.

---

**Table 1**

<table>
<thead>
<tr>
<th>Encoding</th>
<th>Decoding</th>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDP</td>
<td>$2M(k-1)/k$</td>
<td>$2M(k-1)/k$</td>
</tr>
<tr>
<td>Cauchy-RS</td>
<td>$O(\omega M\ell)$</td>
<td>$O(\omega M\ell)$</td>
</tr>
<tr>
<td>Cauchy-RS</td>
<td>$O(m^2)$</td>
<td>Multiplication</td>
</tr>
<tr>
<td>ZD</td>
<td>$mM(k-1)/k$</td>
<td>$O(kM + \frac{mM}{k})$</td>
</tr>
</tbody>
</table>
Table 2
Parameters setting.

<table>
<thead>
<tr>
<th>(k, m) commonly used in today’s systems</th>
<th>(6, 2), (6, 3), (10, 4), (12, 4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(k, m) − 1.33×</td>
<td>(12, 4), (15, 3), (18, 6), (24, 8)</td>
</tr>
<tr>
<td>(k, m) − 1.6×</td>
<td>(12, 7), (15, 9), (18, 10), (24, 14)</td>
</tr>
<tr>
<td>Source file size</td>
<td>1 GB</td>
</tr>
<tr>
<td>Buffer size</td>
<td>600 KB</td>
</tr>
<tr>
<td>Data block size</td>
<td>4 KB</td>
</tr>
</tbody>
</table>

Fig. 6. Performance of RDP codes, ZD codes and Cauchy-RS codes for k = 6, m = 2.

7.1. Encoding and decoding performance comparison

In this subsection we provide the numerical results based on our experiments. All the implementation programs are complied using gcc. Our test platform is a PC with Intel Core i5-2410 CPU running at 2.30 GHz with 8 GB of RAM, and a L1 cache of 32 KB and a L2 cache of 256 KB. We measure the time using the gettimeofday(), which is considered to be the most accurate timing function under Linux operating system. All of our results are the average of 10 runs. In Fig. 6, we show the performance of the encoding and decoding of the three different codes using box plot for the case where k = 6 and m = 2. According to the figure, 10 runs is enough to obtain accurate estimate, as the variance of the ten samples is very small for each of the three codes. For better visual effect, we plot our experimental results using bar chart in other figures.

When it comes to the double-parity case, according to the results shown in Fig. 6, the encoding performance of ZD codes are nearly the same as that of RDP codes, and both RDP and ZD codes outperform Cauchy-RS by roughly 20%. The RDP encoding process requires almost 1.734 GB of XORs, which is actually the same as the XORs required for ZD encoding. This result agrees with the computational complexity analysis in section 5. The encoding time of RDP is even slightly larger than that of ZD codes. This is reasonable because the XOR computing in RDP codes has poorer data address continuity than ZD codes, which may lead to more time for addressing and more cache misses. On the other hand, the decoding performance of ZD codes is not as good as the other two codes. The reason is that we decode one computer word per iteration during the decoding process of ZD codes, which implies large space difference between subsequent memory access. It leads to a lot of cache misses and results in poor performance. That is also the reason for worse decoding performance of ZD codes than that of Cauchy-RS codes when (k, m) = (6, 3), as can be seen from Fig. 8. However, compared with Cauchy-RS codes in the case of (k, m) = (6, 3), the encoding time of ZD codes has been reduced by 24.74%. Also, as shown in Fig. 7 and 8, ZD codes have better performance than Cauchy-RS codes in terms of both encoding and decoding when (k, m) = (10, 4) and (k, m) = (12, 4). We also investigate the number of XORs performed by ZD codes and Cauchy-RS codes. According to Fig. 9, the number of XORs during the encoding process of ZD codes is 20% less than that of Cauchy-RS codes when (k, m) = (6, 2). And the advantage increases to 39.6% when (k, m) = (12, 4). With respect to decoding, as shown in Fig. 10 the number of XORs performed by Cauchy-RS codes is 10.42% more than that performed by ZD codes when (k, m) = (6, 2). In the case of (k, m) = (12, 4), Cauchy-RS codes require 62.04% more XORs required by ZD codes.
Fig. 7. Encoding performance of ZD codes and Cauchy-RS codes for \((k, m) \in \{(6, 3), (10, 4), (12, 4)\}\).

Fig. 8. Decoding performance of ZD codes and Cauchy-RS codes for \((k, m) \in \{(6, 3), (10, 4), (12, 4)\}\).

Fig. 9. Number of XORs performed during encoding process of ZD and Cauchy-RS for \((k, m) \in \{(6, 2), (6, 3), (10, 4), (12, 4)\}\).

Fig. 11 and 12 provide the encoding time and decoding time of the two codes for two different levels of storage overhead, which are 1.33\(\times\) and 1.6\(\times\). As the number of source subfiles increases, the encoding and decoding times of both ZD codes and Cauchy-RS codes are increasing with the growth of \(k\). However, the encoding and decoding times of Cauchy-RS increase much more quickly than those of ZD codes. As the word size \(\omega\) of Cauchy-RS should increase with the growth of \(k + m\). Larger value of \(\omega\) implies more complicated binary encoding matrix for Cauchy-RS and more operations in finite field. Therefore, larger value of \(\omega\) results in worse performance of Cauchy-RS, which is consistent with the result in [3]. And our results show that ZD codes outperforms Cauchy-RS codes at least 45.77\% in encoding time and at least 19.70\% in decoding time when \(\omega = 5\). When \(\omega = 6\), encoding time is reduced by at least 59.87\% and decoding time is reduced by at least 46.93\%. 

Fig. 10. Number of XORs performed during encoding process of ZD and Cauchy-RS for \((k, m) \in \{(6, 2), (6, 3), (10, 4), (12, 4)\}\).
**Fig. 10.** Number of XORs performed during decoding process of ZD and Cauchy-RS for \((k,m) \in \{(6,2), (6,3), (10,4), (12,4)\}\).

**Fig. 11.** Encoding performance of ZD and Cauchy-RS for 1.33\(\times\) and 1.6\(\times\).

**Fig. 12.** Decoding Performance of ZD and Cauchy-RS for 1.3\(\times\) and 1.6\(\times\).
7.2. Storage overhead of ZD codes

As we mentioned, ZD codes introduce a little extra storage overhead to obtain high encoding and decoding performance. The actual value of overhead depends not only on specific code construction, but also the implementation settings. We summarize the proportion of storage overhead based on our analysis and experiment settings in Table 3. Suppose $L'$ is the size of parity block, and $L$ is the size of data block. The overhead is normalized over total storage amount needed for ordinary MDS codes as $\frac{m(L'-1)}{(k+m)L}$.

As shown in the table, the price to pay for ZD codes is negligible, and better performance in encoding and decoding can be obtained in return.

8. Conclusion

A new class of erasure codes, called ZD codes, is designed. Two constructions, which are based respectively on the Vandermonde matrix and the Hankel matrix, are presented. They belong to the class of ZD codes because both of them satisfy the increasing difference property, which is proved to be a sufficient condition for zigzag decodability. The most attractive feature of the ZD codes is that both the encoding and decoding algorithms can be performed in linear time and involve only XOR and bit-shift operations. Such a linear time complexity is lower than those of all existing general-purpose erasure codes that have the combination property. The only price to pay is that the size of parity packets is slightly larger than those of classical MDS codes. This extra overhead, however, is negligible for data storage applications, in which the packet size is typically very large.

To evaluate its practical applicability, we implement the encoding and decoding of the code in C programming language. Based on our implementation, we compare the encoding and decoding performance of ZD codes with RDP codes and Cauchy-RS codes. Since RDP is specially designed for RAID-6, which supports only double parity, it performs better than the other two codes, which are general-purpose codes, as expected. To compare ZD codes with Cauchy-RS codes, a detailed experimental study has been carried out. When the number of total packets $n$, including source packets and parity packets, is less than 16, encoding and decoding of ZD codes is faster than Cauchy-RS codes under the often used parameter settings of practical storage systems. It performs worse than Cauchy-RS codes only when the coding parameters are very small. When $n$ is larger than 16, the encoding performance of ZD codes is far better than Cauchy-RS codes, and the decoding performance of ZD codes is at least 20% better than that of Cauchy-RS codes.

In addition to actual computation time of encoding and decoding, we have also measured the number of XOR operations required by the two codes. Experimental results show that ZD codes always require fewer XOR operations in both encoding and decoding, even if the coding parameters are very small. This result hints that cache miss is the key factor which adversely affects the decoding time of ZD codes. While the actual encoding and decoding times of ZD codes are greater than Cauchy-RS codes for small parameters such as $(k, m) = (6, 3)$, this result depends heavily on today’s computer architecture. The required numbers of XOR operations of the two codes, however, are machine-independent. If the codes are compared not on general-purpose computers, but on specially designed hardware platforms, ZD codes have the potential to outperform Cauchy-RS even for very small coding parameters.

Acknowledgment

This work was partially supported by a grant from the University Grants Committee of the Hong Kong Special Administrative Region, China (Project No. AoE/E-02/08).

Appendix A. Example of zigzag decoding algorithm

Now we illustrate the idea of ZD codes and the associated decoding method as stated in Algorithm 1 by walking through the following example:

Example 1. Let $n = 7, k = 3, m = 4$, and

\[
B(z) = \begin{bmatrix} z^2 & z & 1 \\ z & 1 & 1 \\ 1 & 1 & z \\ 1 & z & z^2 \end{bmatrix}.
\]
The seven coded packets are graphically shown in Fig. 13. Now we explain how zigzag decoding algorithm works, assuming that the three parity packets, $c_5$, $c_6$, and $c_7$, are used for decoding.

The three input parity packets for decoding, $y_1$, $y_2$ and $y_3$ are shown in Fig. 14. The $k \times k$ input integer array, $T$, is given by

$$T = \begin{bmatrix}
1 & 0 & 0 \\
0 & 0 & 1 \\
0 & 1 & 3
\end{bmatrix}. \quad (A.1)$$
whose $(i, j)$-th entry, where $1 \leq i, j \leq 3$, represents the amount of bit-offset of source packet $j$ in the $i$-th packet for decoding.

The first step of the algorithm is to initialize the variable matrix $B$ as $T$. Note that the $i$-th row of this variable matrix records the offset of the first not-yet-decoded bit of the source packets in the $i$-th coded packet for decoding. Then from Step 2 to Step 9, we compute $k$ integer arrays $A_i$ for each $y_i$, where $A_i[j]$ indicates how many source packets are XORed together to obtain the $j$-th bit of $y_i$. The initial value of $A_i$ is given under each coded packet in Fig. 14. Besides, the variable $p_i$ in Step 4 represents the first not-yet-decoded bit in the $i$-th coded packet for decoding. All the $p_i$s are initialized to 1.

The decoding procedure is carried out in the for-loop starting from Step 10. In Step 11, the lowest-indexed coded packet with an exposed bit is identified. Consider the first iteration, the smallest $i^*$ such that $A_i[p_{i^*}] = 1$ is 3, implying $A_3$ has an exposed bit while $A_1$ and $A_2$ do not. Now we have to find which source packets contribute to this exposed bit. It is performed by finding the smallest element in the $i^*$-th row of $B$ and returning the position. In this case, the smallest element is $B[3][j^*]$, where $j^* = 1$. Therefore, $s_{j^*} = s_1$ is the source packet that contributes to this exposed bit. Note that the offset of source packet $s_1$ in encoding $y_3$ is $T[3][1] = 0$. Thus it must be the bit of position $h := p_{j^*} - T[3][1] = 1$ in $s_1$ that was enrolled in the exposed bit, and the value of the exposed bit is $b := Y_3[p_{j^*}] = Y_3[1]$, as shown in Step 13. Now we may conclude $X_1[h] = X_1[1] := b$, as shown in Step 14. $X_1[1]$ in fact is equal to the first bit of $s_1$, i.e., $s_{1,0}$.

Before entering into the next iteration, we need to update some variables, which are done in Steps 15–26. The bit decoded out should be subtracted from the three encoded packets. It is executed by the following XOR operations in Step 16:

$$Y_1[2] := Y_1[2] \oplus b,$$
$$Y_2[1] := Y_2[1] \oplus b,$$

The matrix $B$, which records the offset of the first not-yet-decoded bit of source packets, is updated by adding 1 to each element in its $j^*$-th column, i.e., the first column in our example:

$$B := B + \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 2 & 0 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & 3 \end{bmatrix}.$$ 

Also the $k$ binary arrays which indicate the number of overlapping source packets should be updated as follows:

$$A_1 := [2 \ 2 \ 3 \ \cdots \ 3 \ 2 \ 1 \ 0 \ 0];$$
$$A_2 := [1 \ 2 \ 3 \ \cdots \ 3 \ 1 \ 0 \ 0];$$
$$A_3 := [0 \ 2 \ 2 \ \cdots \ 3 \ 3 \ 2 \ 1 \ 1].$$

The value of $p_3$ becomes 2, meaning that next time we start from $A_3[2]$ to find the smallest $i^*$ in Step 11. In other words, $p_i$ always points to the first non-zero element of $A_i$.

In this example, it is easy to see that the sequence of coded packets that have “exposed” bits is 3, 2, 1, 3, 2, 1, ..., which is periodic. This feature can be exploited to further speed up the decoding algorithm.

References


