Shift-XOR Storage Codes with Two-tone Generator Matrices: Design and Implementation

***Abstract*—We introduce a new class of generator matrices, called two-tone matrices, for storage codes using shift and XOR operations. Compared with the existing shift-XOR storage codes with increasing-difference generator matrices, our codes only have $1/3$ to $1/2$ storage overhead for practical cases. We propose a bandwidth-overhead-free, in-place decoding algorithm for two-tone shift-XOR storage codes, which requires the same number of XOR operations for generating the corresponding coded subsequences. The two-tone shift-XOR storage codes have lower encoding/decoding computation complexity than Reed-Solomon codes and several MDS codes based on cyclic-shift and XOR operations. To verify the practical performance, we implement two-tone shift-XOR storage codes using C++ and compare the encoding/decoding throughput with the state-of-the-art implementation of Reed-Solomon codes. For certain practical cases, our codes can achieve from 40% to 80% higher encoding/decoding throughput than that of Reed-Solomon codes.**

I. Introduction

Erasure coding is an efficient approach to ensure the reliability of data stored in distributed storage systems. Reed-Solomon (RS) codes [1] are widely used MDS (maximum-distance separable) codes in distributed storage systems, such as HDFS-RAID in Facebook [2] and GFS II in Google [3], [4].

The encoding and decoding of RS codes are over finite fields, which entails high computational costs. By using Cauchy matrices instead of Vandermonde matrices as in RS codes, Cauchy RS codes [5] were proposed. The multiplication of two elements in $GF(2^m)$ can be represented as a binary matrix-vector multiplication. For particular Cauchy matrices, the binary matrices corresponding to the matrix entries are relatively sparse so that the computational cost of field multiplication in Cauchy RS codes can be reduced [6]. A storage code that enables correct decoding from any k out of n storage nodes is called an $[n, k]$ code. With a sufficiently large finite field, both $[n, k]$ RS codes and $[n, k]$ Cauchy RS codes exist for any $k \leq n$.

To achieve lower computational complexity, cyclic shift over finite fields and XOR operations were employed in array codes [7]. For any $k \leq n$, MDS $[n, k]$ erasure codes using cyclic shift and XOR operations have been constructed using Vandermonde-type generator matrices [8], [9]. Another coding scheme using cyclic shift and integer addition was proposed in [10].

In this paper, we focus on storage codes which employs (noncyclic) shift and XOR operations, which potentially have the lowest encoding and decoding computational costs among the existing $[n, k]$ storage coding techniques for any valid $k \leq n$. Shift-XOR-based regenerating codes were also proposed for similar purposes [11], [12]. Sung and Gong [13] proposed $[n, k]$ storage codes for any $k \leq n$ using shift and XOR operations, where the generator matrix satisfies the *increasing difference property*. In an $[n, k]$ shift-XOR code, a data file formed by k message sequences each of L bits is encoded into n sequences, where encoding one coded sequences requires at most $(k - 1)L$ XOR operations. Using a ZigZag decoding algorithm, the data file can be decoded from any k out of the n coded sequences. However, each coded sequence may have more than L bits because of the shift operations, thus the shift-XOR codes are not strictly MDS.

For an $[n, k]$ shift-XOR code, each coded sequence may be longer than L bits, and hence two kinds of overheads are generated: First, the total number of storage bits minus nL is called the *storage overhead*. Second, the total number of bits retrieved for decoding minus kL is called the *bandwidth overhead*. In a series of papers by Fu, Xiao and Yang [12], [14], [15], the increasing difference property was refined to reduce the storage overhead, and a decoding algorithm called *shift-XOR elimination* was proposed. From any k out of the n coded sequences, the shift-XOR elimination retrieves subsequences of exactly kL bits for decoding, and hence has zero bandwidth overhead. Moreover, the number of XOR operations for the decoding algorithm and generating the *subsequences* are the same, which is upper bounded by $k(k-1)L$ XOR operations. This implies the shift-XOR elimination costs less decoding time than the ZigZag decoding algorithm. In addition, the shift-XOR elimination is in-place implementable in the sense that the output and the input sequences can share the same storage space without any auxiliary space for caching intermediate results.

Towards practical distributed storage system applications, in this paper, we study shift-XOR codes in terms of both coding design and implementation. First, we propose a more general class of generator matrices for shift-XOR codes, called *two-tone matrices*, together with a decoding algorithm that is as efficient as the shift-XOR elimination. Second, We implement the two-tone shift-XOR codes using C++ and compare the encoding and decoding throughputs with the state-of-the-art implementations of RS codes and Cauchy RS codes to demonstrate the potential advantages of shift-XOR codes in real-world.

In particular, using Refined Increasing Difference (RID) generator matrices [15], the numbers of bit shift of the message sequences follow an increasing order for each coded sequence. Our two-tone generator matrices generalize RID ones by allowing the numbers of bit shift decreasing for certain coded sequences, and hence can reduce the storage overhead. Compared with the Vandermonde RID generator matrices, which have the smallest storage overhead among RID ones, the corresponding two-tone generator matrices can reduce up to half of the storage overhead. Moreover, based on two-tone matrices, we propose systematic shift-XOR storage codes that can further reduce the storage overheads to less than 10% of that of the non-systematic codes for some practical parameters. For decoding, we generalize the shift-XOR elimination to handle both the increasing and decreasing order of the numbers of bit shift of the message sequences. Our algorithm preserves the advantages of the shift-XOR elimination, including in-place and bandwidth-overhead free.

Due to the significant progress in design and analysis of shift-XOR codes, a study of the practical performance of shift-XOR codes is desired. In existing literature [16], only shift-XOR codes with ZigZag decoding have been implemented and compared with the Cauchy RS codes implemented in the Jerasure library [17]. Their results showed that shift-XOR codes outperformed the Cauchy RS codes in both encoding and decoding time. Depending on the word size and the values of n and k, the time reduction varied from 20% to 60%. The wide applications of storage codes facilitated the recent emergence of new coding libraries like Longhair (for Cauchy RS codes) [18] and ISA-L (for RS codes) [19], which both demonstrate superior performance than the Jerasure library [17] while not yet used as a comparison in literature.

In this paper, we implement the systematic two-tone shift-XOR storage codes using C++, and compare the single-core encoding/decoding throughput of systematic RS codes (implemented in ISA-L and Jerasure) and Cauchy RS codes (implemented in Longhair). Our codes achieve from 40 to 80 percent higher encoding/decoding throughput than ISA-L, which is considered the state-of-the-art encoding/decoding library. Compared with Cauchy-RS codes implemented in the Longhair library, our codes can achieve 80 percent higher encoding/decoding throughput for small file size (128KB) and from 5 to 8 times the encoding/decoding throughput for large file size (512MB). Compared with the RS codes implemented in Jerasure library, our codes can achieve 10 times the encoding/decoding throughput.

The remainder of the paper is organized as follows. In Section II, we introduce shift-XOR codes with two-tone generator matrices, and discuss an algorithm for solving a shift-XOR system. In Section III, we construct two-tone shift-XOR codes with the minimal storage overhead, including a systematic construction. In Section IV, we implement our codes on a single PC and compare with some major coding libraries.

II. Shift-XOR Codes with Two-tone Generator Matrices

In this section, we introduce the shift-XOR codes with two-tone generator matrices, together with the decoding algorithm. We denote a range of integers from i to j by $i : j$. When $i > j$, $i : j$ is the empty set. For a binary sequence $\mathbf{a}$, denoted by bold lowercase letters, the i-th entry is denoted by $\mathbf{a}[i]$. The subsequence of $\mathbf{a}$ from the i-th entry to the j-th entry is denoted by $\mathbf{a}[i : j]$. For a sequence $\mathbf{a}$ of length L, we use the convention that $\mathbf{a}[l] = 0$ for $l < 0$ and $l > L$.

For a sequence $\mathbf{a}$ of L bits and a natural number $t \geq 0$, the shift operator z^t pads t zeros in front of $\mathbf{a}$, so that the $(l+t)$-th entry of $z^t\mathbf{a}$ is equal to the l-th of $\mathbf{a}$, i.e., for $l = 1, \dots, L+t$,

$$\left(z^t\mathbf{a}\right)[l] = \mathbf{a}[l-t].$$

Let $\mathbf{a}_1$ and $\mathbf{a}_2$ be two sequences of length L_1 and L_2, respectively. Their addition $\mathbf{a}_1 + \mathbf{a}_2$ is bit-wise exclusive-or (XOR). If these two sequences are not of the same length, zeros are appended after the shorter one before the addition so that for $l = 1, \dots, \max\{L_1, L_2\}$,

$$\left(\mathbf{a}_1 + \mathbf{a}_2\right)[l] = \mathbf{a}_1[l] \oplus \mathbf{a}_2[l].$$

A. Shift-XOR Codes

Consider k binary message sequences, each of L bits, where the j-th sequence is denoted as $\mathbf{x}_j$. The *generator matrix* used to encode the message sequences is an $n \times k$ matrix $\mathbf{\Psi} = (z^{t_{i,j}})$, where $t_{i,j} \geq 0$ determines the number of bit shifts of the j-th message sequence in the i-th *coded sequence* $\mathbf{y}_i$, where $\mathbf{y}_i$ takes the following form:

$$\mathbf{y}_i = \sum_{j=1}^{k} z^{t_{i,j}}\mathbf{x}_j, \ \forall 1 \leq i \leq n.$$

Denote $\mathbf{Y} = (\mathbf{y}_1, \mathbf{y}_2, \dots, \mathbf{y}_k)^\top$. The encoding can be written in the matrix form

$$\mathbf{Y} = \mathbf{\Psi}\mathbf{X}, \tag{1}$$

which is also called an $n \times k$ shift-XOR system.

Existing works have studied the generator matrices satisfying the (refined) increasing difference properties [12], [13], [15], for which an efficient decoding algorithm exists. Here we introduce a more general class of generator matrices, called *two-tone* matrices.

Definition 1 (Two-tone Matrix)**.** An $n \times k$ matrix $\mathbf{\Psi} = (z^{t_{i,j}})$ is said to be *two-tone* if for certain integer $0 \leq d \leq n$ the following conditions hold:

1) for any $1 \leq i_1 < i_2 \leq n$, $1 \leq j_1 < j_2 \leq k$, $t_{i_1,j_2} - t_{i_1,j_1} < t_{i_2,j_2} - t_{i_2,j_1}$;
2) for any $1 \leq i \leq d$, $1 \leq j_1 < j_2 \leq k$, $t_{i,j_2} - t_{i,j_1} \leq 0$;
3) for any $d < i \leq n$, $1 \leq j_1 < j_2 \leq k$, $t_{i,j_2} - t_{i,j_1} > 0$.

Here d is called the *divide* of $\mathbf{\Psi}$.

By condition 2) of the definition, if $t_{i,j_2} - t_{i,j_1} = 0$ for certain $j_1 < j_2$, then $i = d$. In other words, only the divide row can have zero differences. It is easy to verify that any submatrix of a two-tone matrix is still a two-tone matrix.

The increasing difference matrices [13] and the refined increasing difference matrices [12], [15] are special cases of two-tone matrices: the increasing difference property corresponds to $d = 0$ and the Refined Increasing Difference (RID) property corresponds to $d = 1$ in two-tone matrices. Different from the (refined) increasing difference properties, two-tone matrices allow the numbers of bit shifts in a row following a descending order.

B. Decoding Examples

We first use examples to illustrate how to solve two-tone shift-XOR systems. As our algorithm is based on the shift-XOR elimination, which was proposed for RID systems [12], we first use an example to explain how does it work.

Example 1. Consider the 2×2 shift-XOR system

$$\begin{bmatrix} \mathbf{y}_1 \\ \mathbf{y}_2 \end{bmatrix} = \begin{bmatrix} 1 & z \\ 1 & z^2 \end{bmatrix} \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{bmatrix}. \tag{2}$$

As the generator matrix is RID, the shift-XOR elimination can be applied to solve the system using subsequences of $\mathbf{y}_1, \mathbf{y}_2$:

$$\hat{\mathbf{x}}_1 = \mathbf{y}_2[1:L],$$
$$\hat{\mathbf{x}}_2 = \mathbf{y}_1[2:L+1].$$

Expanding the shift operator, we get for $1 \le l \le L$,

$$\hat{\mathbf{x}}_1[l] = \mathbf{x}_1[l] + \mathbf{x}_2[l-2],$$
$$\hat{\mathbf{x}}_2[l] = \mathbf{x}_2[l] + \mathbf{x}_1[l+1].$$

The system is solved by multiple iterations. An iteration index ℓ is initialized as 1, and is increased by 1 after each iteration. The following processes are done in each iteration:

- When $\ell = 1$, $\mathbf{x}_1[1] = \hat{\mathbf{x}}_1[1]$ is solved.
- For each iteration $\ell = 2, 3, \ldots, L+1$, $\mathbf{x}_1[\ell]$ and $\mathbf{x}_2[\ell-1]$ are solved sequentially by equations
$$\mathbf{x}_1[\ell] = \hat{\mathbf{x}}_1[\ell] + \mathbf{x}_2[\ell-2],$$
$$\mathbf{x}_2[\ell-1] = \hat{\mathbf{x}}_2[\ell-1] + \mathbf{x}_1[\ell],$$
respectively, where we can check inductively that all the message bits used on the RHS have been solved previously.

Example 2. The 3×3 system

$$\begin{bmatrix} \mathbf{y}_1 \\ \mathbf{y}_2 \\ \mathbf{y}_3 \end{bmatrix} = \begin{bmatrix} z^4 & z^2 & 1 \\ z^2 & z & 1 \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \mathbf{x}_3 \end{bmatrix} \tag{3}$$

does not have a RID generator matrix, but it is equivalent to one with a RID generator matrix:

$$\begin{bmatrix} \mathbf{y}_3 \\ \mathbf{y}_2 \\ \mathbf{y}_1 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & z & z^2 \\ 1 & z^2 & z^4 \end{bmatrix} \begin{bmatrix} \mathbf{x}_3 \\ \mathbf{x}_2 \\ \mathbf{x}_1 \end{bmatrix}.$$

Therefore, system (3) can also be solved by the shift-XOR elimination.

In general, for a $k \times k$ two-tone matrix with divide k, by reversing the order of rows and columns, we obtain a two-tone matrix with divide 1, i.e., an RID matrix.

Example 3. Next, we consider a more general example:

$$\begin{bmatrix} \mathbf{y}_1 \\ \mathbf{y}_2 \\ \mathbf{y}_3 \\ \mathbf{y}_4 \\ \mathbf{y}_5 \end{bmatrix} = \begin{bmatrix} z^8 & z^6 & z^4 & z^2 & 1 \\ z^4 & z^3 & z^2 & z & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & z & z^2 & z^3 & z^4 \\ 1 & z^2 & z^4 & z^6 & z^8 \end{bmatrix} \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \mathbf{x}_3 \\ \mathbf{x}_4 \\ \mathbf{x}_5 \end{bmatrix},$$

where the generator matrix has divide 3. The system can be further written as two systems of the same set of variables:

$$\begin{bmatrix} \mathbf{y}_1 \\ \mathbf{y}_2 \\ \mathbf{y}_3 \end{bmatrix} = \begin{bmatrix} z^4 & z^2 & 1 \\ z^2 & z & 1 \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} \mathbf{x}_3 \\ \mathbf{x}_4 \\ \mathbf{x}_5 \end{bmatrix} + \begin{bmatrix} z^8 & z^6 \\ z^4 & z^3 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{bmatrix}, \tag{4}$$

$$\begin{bmatrix} \mathbf{y}_4 \\ \mathbf{y}_5 \end{bmatrix} = \begin{bmatrix} 1 & z \\ 1 & z^2 \end{bmatrix} \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{bmatrix} + \begin{bmatrix} z^2 & z^3 & z^4 \\ z^4 & z^6 & z^8 \end{bmatrix} \begin{bmatrix} \mathbf{x}_3 \\ \mathbf{x}_4 \\ \mathbf{x}_5 \end{bmatrix}. \tag{5}$$

We observe that without the second term on the RHS, both systems above can be solved by the shift-XOR elimination (ref. (2) and (3)). The second term in each system can be regarded as the interference from other one. Our idea is to solve (4) and (5) using two individual shift-XOR eliminations, together with interference cancellation between each other.

Same as the shift-XOR elimination, we define subsequences

$$\begin{aligned} \hat{\mathbf{x}}_1 &= \mathbf{y}_5[1:L], & \hat{\mathbf{x}}_3 &= \mathbf{y}_3[1:L], \\ \hat{\mathbf{x}}_2 &= \mathbf{y}_4[2:L+1], & \hat{\mathbf{x}}_4 &= \mathbf{y}_2[2:L+1], \\ & & \hat{\mathbf{x}}_5 &= \mathbf{y}_1[1:L], \end{aligned}$$

where $\hat{\mathbf{x}}_1, \hat{\mathbf{x}}_2$ are used by solving (5), and $\hat{\mathbf{x}}_3, \hat{\mathbf{x}}_4, \hat{\mathbf{x}}_5$ are used by solving (4). Expanding the shift operator, we further have for $1 \le l \le L$,

$$\hat{\mathbf{x}}_1[l] = \mathbf{x}_1[l] + \mathbf{x}_2[l-2] + \mathbf{x}_3[l-4] + \mathbf{x}_4[l-6] + \mathbf{x}_5[l-8],$$
$$\hat{\mathbf{x}}_2[l] = \mathbf{x}_2[l] + \mathbf{x}_1[l+1] + \mathbf{x}_3[l-1] + \mathbf{x}_4[l-2] + \mathbf{x}_5[l-3],$$
$$\hat{\mathbf{x}}_3[l] = \mathbf{x}_3[l] + \mathbf{x}_4[l] + \mathbf{x}_5[l] + \mathbf{x}_1[l] + \mathbf{x}_2[l],$$
$$\hat{\mathbf{x}}_4[l] = \mathbf{x}_4[l] + \mathbf{x}_3[l-1] + \mathbf{x}_5[l+1] + \mathbf{x}_1[l-3] + \mathbf{x}_2[l-2],$$
$$\hat{\mathbf{x}}_5[l] = \mathbf{x}_5[l] + \mathbf{x}_3[l-4] + \mathbf{x}_4[l-2] + \mathbf{x}_1[l-8] + \mathbf{x}_2[l-6].$$

We use ℓ^+ and ℓ^- as the iteration indices of system (5) and (4), respectively. In the first iteration of both systems, we see that some bits can be solved directly:

- For (5), we solve $\mathbf{x}_1[1] = \hat{\mathbf{x}}_1[1]$.
- For (4), we solve $\mathbf{x}_5[1] = \hat{\mathbf{x}}_5[1]$.

Now, $\ell^+ = \ell^- = 2$. The following operations are performed for each following iteration:

- (System (5)) Solve $\mathbf{x}_1[\ell^+]$ and $\mathbf{x}_2[\ell^+ - 1]$ sequentially using $\hat{\mathbf{x}}_1$ and $\hat{\mathbf{x}}_2$, respectively, together with the previously solved bits.
- (System (4)) Solve $\mathbf{x}_5[\ell^-]$, $\mathbf{x}_4[\ell^- - 1]$, $\mathbf{x}_3[\ell^- - 1]$ sequentially using $\hat{\mathbf{x}}_5, \hat{\mathbf{x}}_4, \hat{\mathbf{x}}_3$, respectively, together with the previously solved bits.
- Increase the iteration indices ℓ^+ and ℓ^- by 1.

C. Two-tone Shift-XOR Elimination

Now we consider the $k \times k$ system of shift-XOR equations

$$\begin{bmatrix} \mathbf{y}_1 & \mathbf{y}_2 & \cdots & \mathbf{y}_k \end{bmatrix}^\top = \mathbf{\Psi} \begin{bmatrix} \mathbf{x}_1 & \mathbf{x}_2 & \cdots & \mathbf{x}_k \end{bmatrix}^\top, \tag{6}$$

where $\mathbf{\Psi} = (z^{t_{i,j}})$ is a two-tone matrix with divide d. We give an algorithm to solve the above general system when $\mathbf{y}_1, \ldots, \mathbf{y}_k$ are given. Our algorithm generalizes the shift-XOR elimination in [12], and is called *two-tone elimination*.

The system (6) can be written as two sub-systems:

$$\begin{bmatrix}\mathbf{y}_1 & \mathbf{y}_2 & \cdots & \mathbf{y}_d\end{bmatrix}^\top = \mathbf{\Psi}^- \begin{bmatrix}\mathbf{x}_1 & \cdots & \mathbf{x}_k\end{bmatrix}^\top, \tag{6^-}$$

$$\begin{bmatrix}\mathbf{y}_{d+1} & \mathbf{y}_{d+2} & \cdots & \mathbf{y}_k\end{bmatrix}^\top = \mathbf{\Psi}^+ \begin{bmatrix}\mathbf{x}_1 & \cdots & \mathbf{x}_k\end{bmatrix}^\top. \tag{6^+}$$

Define for $u = 1, 2, \dots, k$ the subsequence

$$\hat{\mathbf{x}}_u = \mathbf{y}_{k+1-u}[(t_{k+1-u,u}+1) : (t_{k+1-u,u}+L)].$$

Substituting into (6) and expanding the shift operator, we have

$$\hat{\mathbf{x}}_u[l] = \mathbf{x}_u[l] + \sum_{j \neq u} \mathbf{x}_j[l - t_{k+1-u,j} + t_{k+1-u,u}],$$

where we see that $\hat{\mathbf{x}}_u$, $u = 1, \dots, k$ involve all the bits we want to decode.

These two sub-systems are solved by two modified shift-XOR elimination interactively, where $\mathbf{x}_{k-d+1}, \dots, \mathbf{x}_k$ are variables for (6^-), and $\mathbf{x}_1, \dots, \mathbf{x}_{k-d}$ are variables for (6^+). The two sub-systems are solved by multiple iterations. We use ℓ^+ and ℓ^- as the iteration indices of system (6^+) and (6^-), respectively. Initially, $\ell^+ = \ell^- = 1$. After each iteration in each sub-system, the corresponding index is increased by 1. The operations of each iteration depend on the value of ℓ^+ and ℓ^-. We define the following parameters that can help us to separate iterations into segments:

$$T_i^+ = t_{k-i,i+1} - t_{k-i,i}, \ 1 \leq i < k-d,$$
$$T_i^- = t_{i+1,i} - t_{i+1,i+1}, \ 1 \leq i < d.$$

Define $T_{i:j}^+ = \sum_{l=i}^{j} T_l^+$ and $T_{i:j}^- = \sum_{l=i}^{j} T_l^-$.

For a number of iterations at the beginning, both sub-systems can be solved separately with the following operations respectively for each iteration:

- For $b = 1, 2, \dots, k-d-1$, for each iteration ℓ^+ in $T_{1:b-1}^+ + (1 : T_b^+)$, solve $\mathbf{x}_u[\ell^+ - T_{1:u-1}^+]$ sequentially for $u = 1, 2, \dots, b$.
- For $b = 1, 2, \dots, d-1$, for each iteration ℓ^- in $T_{1:b-1}^- + (1 : T_b^-)$, solve $\mathbf{x}_{k-u+1}[\ell^- - T_{1:u-1}^-]$ sequentially for $u = 1, 2, \dots, b$.

In the above process, one bit is solved implies that it is back substituted into the subsequences it involves in. After the above iterations, $\ell^+ = T_{1:k-d-1}^+ + 1$ and $\ell^- = T_{1:d-1}^- + 1$. Then the following operations are performed sequentially for each iteration:

- Solve $\mathbf{x}_u[\ell^+ - T_{1:u-1}^+]$ sequentially using $\hat{\mathbf{x}}_u$ and previously solved bits, for $u = 1, 2, \dots, k-d$.
- Solve $\mathbf{x}_{k-u+1}[\ell^- - T_{1:u-1}^-]$ sequentially using $\hat{\mathbf{x}}_{k-u+1}$ and previously solved bits, $u = 1, 2, \dots, d$.

The algorithm stops after all the bits are solved.

Same as the shift-XOR elimination, we can show that the two-tone elimination can be implemented in-place, i.e., no auxiliary space is required to storage the intermediate shift-XOR results. A pseudocode is shown in Algorithm 1 to demonstrate an in-place implementation of two-tone elimination. Similar to the analysis of shift-XOR elimination, the two-tone elimination needs less than $k(k-1)$ XOR operations and $O(k^2L)$ integer operations. The correctness of the two-tone elimination can be guaranteed by the following Theorem.

Algorithm 1 Two-tone elimination with in-place implementation

```
Input: bits stored in sequences $\hat{\mathbf{x}}_u$, $u = 1, 2, \dots, k$.
Output: decoded message, stored in $\hat{\mathbf{x}}_u$.
 1: Initialize $\ell^+ \leftarrow 0$ and $\ell^- \leftarrow 0$;
 2: for $b \leftarrow 1, 2, \dots, k-d-1$ do
 3:   for $T_b^+$ iterations do
 4:     $\ell^+ \leftarrow \ell^+ + 1$;
 5:     for $u \leftarrow 1 : b$ do
 6:       $l \leftarrow \ell^+ - T_{1:u-1}^+$;
 7:       for $v \leftarrow 1, \dots, u-1, u+1, k$ do
 8:         $\hat{\mathbf{x}}_v[t_{k-v+1,u} - t_{k-v+1,v} + l] \leftarrow \hat{\mathbf{x}}_v[t_{k-v+1,u} - t_{k-v+1,v} + l] \oplus \hat{\mathbf{x}}_u[l]$;
 9: for $b \leftarrow 1, 2, \dots, d-1$ do
10:   for $T_b^-$ iterations do
11:     $\ell^- \leftarrow \ell^- + 1$
12:     for $u \leftarrow 1, 2, \dots, b$ do
13:       $l \leftarrow \ell^- - T_{1:u-1}^-$
14:       for $v \leftarrow 1, \dots, u-1, u+1, k$ do
15:         $\hat{\mathbf{x}}_v[t_{k-v+1,k-u+1} - t_{k-v+1,v} + l] \leftarrow \hat{\mathbf{x}}_v[t_{k-v+1,k-u+1} - t_{k-v+1,v} + l] \oplus \hat{\mathbf{x}}_{k-u+1}[l]$;
16: for $L$ iterations do
17:   $\ell^+ \leftarrow \ell^+ + 1$
18:   for $u \leftarrow 1 : k-d$ do
19:     $l \leftarrow \ell^+ - T_{1:u-1}^+$
20:     for $v \leftarrow 1, \dots, u-1, u+1, k$ do
21:       $\hat{\mathbf{x}}_v[t_{k-v+1,u} - t_{k-v+1,v} + l] \leftarrow \hat{\mathbf{x}}_v[t_{k-v+1,u} - t_{k-v+1,v} + l] \oplus \hat{\mathbf{x}}_u[l]$;
22:   $\ell^- \leftarrow \ell^- + 1$
23:   for $u \leftarrow 1, 2, \dots, d$ do
24:     $l \leftarrow \ell^- - T_{1:u-1}^-$
25:     for $v \leftarrow 1, \dots, u-1, u+1, k$ do
26:       $\hat{\mathbf{x}}_v[t_{k-v+1,k-u+1} - t_{k-v+1,v} + l] \leftarrow \hat{\mathbf{x}}_v[t_{k-v+1,k-u+1} - t_{k-v+1,v} + l] \oplus \hat{\mathbf{x}}_{k-u+1}[l]$;
```

Theorem 1. *Consider a $k \times k$ system of shift-XOR equations $(\mathbf{y}_1 \ \cdots \ \mathbf{y}_k)^\top = \mathbf{\Psi}(\mathbf{x}_1 \ \cdots \ \mathbf{x}_k)^\top$ with $\mathbf{\Psi}$ being a two-tone matrix. The two-tone elimination can successfully decode $\mathbf{x}_u$, $u = 1, \dots, k$ using*

$$\hat{\mathbf{x}}_u = \mathbf{y}_{k+1-u}[(t_{k-u+1,u}+1) : (t_{k-u+1,u}+L)].$$

Theorem 1 can be verified by expressing each bit to decode using $\hat{\mathbf{x}}_u$, $u = 1, 2, \dots, k$ and the previously decoded bits. The proof is omitted due to the page limit.

D. Decoding an $[n, k]$ Storage Code

Now we consider a storage system of n storage nodes employing an $[n, k]$ shift-XOR code as defined in (1) with a two-tone generator matrix. The n coded sequences are stored at n distinct storage nodes. We show that the file can be decoded from any k out of the n storage nodes.

Assume that the decoder has access to k nodes with the indices in descending order, i.e., $i_1 > i_2 > \cdots > i_k$. As any submatrix of a two-tone generator matrix is also a two-tone matrix, the k coded sequences that can be accessed by the decoder form a $k \times k$ two-tone shift-XOR system, which can be solved using the two-tone elimination.

Our decoding scheme consists of two stages: the transmission stage and the decoding stage. In the transmission stage, node i_u transmits $\mathbf{y}_{i_u}$ with the range of $[(t_{i_u,u}+1):(t_{i_u,u}+L)]$ to the decoder and stores in $\hat{\mathbf{x}}_u$, $u=1,2,\dots,k$, i.e., $\hat{\mathbf{x}}_u = \mathbf{y}_{i_u}[(t_{i_u,u}+1):(t_{i_u,u}+L)]$. As each node transmits exactly L bits to the decoder, the decoding scheme has no bandwidth overhead. In the decoding stage, the decoder applies the two-tone elimination on $\hat{\mathbf{x}}_u$, $u=1,2\dots,k$. Then $\hat{\mathbf{x}}_u$ can be decoded into $\mathbf{x}_u$ in-place.

III. Storage Overhead Optimized Generator Matrices

Consider a storage system of n storage nodes employing shift-XOR codes described in the previous section. Due to the shift operation, each storage node may store more than L bits, the length of a message sequence. Each coded sequence $\mathbf{y}_i$ has $L+\max_{j=1}^{k} t_{i,j}$ bits, and hence the total number of bits stored at n codes is $nL+\sum_{i=1}^{n}\max_{j=1}^{k} t_{i,j}$. The *storage overhead* of the shift-XOR codes with generator matrix $\mathbf{\Psi}=(z^{t_{i,j}})$ is defined as

$$S(\mathbf{\Psi}) = \sum_{i=1}^{n} \max_{j=1}^{k} t_{i,j}. \tag{7}$$

In this section, we give specific constructions of two-tone matrices to minimize the storage overhead.

A. Reflected Vandermonde Matrices

We define a special class of two-tone matrices that generalize the Vandermonde matrices.

Definition 2 (Two-tone and Reflected Vandermonde Matrices)**.** The $n \times k$ two-tone Vandermonde matrix $\mathbf{\Psi}=(z^{t_{i,j}})$ with divide d is defined as

$$t_{i,j} = \begin{cases} (d-i)(k-j), & 1 \le i \le d, \\ (i-d)(j-1), & d < i \le n. \end{cases} \tag{8}$$

Further, $\mathbf{\Psi}$ is called a *reflected Vandermonde matrix* if

$$d = \begin{cases} \frac{n+1}{2}, & n \text{ is odd}, \\ \frac{n}{2}+1 \text{ or } \frac{n}{2}, & n \text{ is even}. \end{cases}$$

It is easy to check that the generator matrix defined by (8) is a two-tone matrix. When $d=1$, a two-tone Vandermonde matrix becomes a usual Vandermonde matrix. The reflected Vandermonde matrix can achieve minimal storage overhead among all two-tone ones of the same size, proved in the next theorem.

Theorem 2. *The storage overhead of an $[n,k]$ shift-XOR code with two-tone generator matrix is lower bounded by $\frac{(n^2-1)(k-1)}{4}$ when n is odd, and $\frac{n^2(k-1)}{4}$ when n is even, and the lower bound is achieved if and only if the generator matrix is the reflected Vandermonde matrix.*

Proof: For an $n \times k$ two-tone generator matrix $\mathbf{\Psi}$ with divide d, following the definition in (7)

$$\begin{aligned} S(\mathbf{\Psi}) &= \sum_{i=1}^{d} t_{i,1} + \sum_{i=d+1}^{n} t_{i,k} \\ &\ge \sum_{i=1}^{d}(t_{i,1}-t_{i,k}) + \sum_{i=d+1}^{n}(t_{i,k}-t_{i,1}) \\ &\ge \sum_{i=1}^{d}(d-i)(k-1) + \sum_{i=d+1}^{n}(i-d)(k-1) \\ &= \frac{k-1}{2}\left(2d^2-2(n+1)d+n^2+n\right), \end{aligned}$$

where the first inequality follows from $t_{i,1} \ge 0$ for $1 \le i \le d$ and $t_{i,k} \ge 0$ for $d < i \le n$; the second inequality follows from a property of two-tone matrices proved in Lemma 1 in Appendix. To guarantee the equalities in above two inequalities, the sufficient and necessary condition is

$$\begin{cases} t_{i,k}=0, t_{i,a}-t_{i,a+1}=d-i, & 1 \le i \le d, \\ t_{i,1}=0, t_{i,a+1}-t_{i,a}=i-d, & d < i \le n. \end{cases}$$

This condition is equivalent to (8).

- When n is odd, we have $S(\mathbf{\Psi}) \ge \frac{(n^2-1)(k-1)}{4}$, where the equality holds if and only if $d=\frac{n+1}{2}$, which corresponds to the first case of definition of the reflected Vandermonde matrix.
- When n is even, we have $S(\mathbf{\Psi}) \ge \frac{n^2(k-1)}{4}$, where the equality holds if and only if $d=\frac{n}{2}$ or $d=\frac{n}{2}+1$, which corresponds to the second case of definition of the reflected Vandermonde matrix.

Combined together, the proof is completed. ■

B. Systematic Two-tone Storage Codes

To further reduce the storage overhead, we give the construction of *systematic* two-tone storage codes. An $[n,k]$ *systematic* two-tone shift-XOR code has the generator matrix

$$\mathbf{\Psi} = \begin{bmatrix} \mathbf{I} \\ \mathbf{\Phi} \end{bmatrix}, \tag{9}$$

where $\mathbf{I}$ is the $k \times k$ identity matrix and $\mathbf{\Phi}$ is an $(n-k)\times k$ two-tone matrix. Note that the first k coded sequences are identical to the message sequences, i.e., $\mathbf{y}_i = \mathbf{x}_i$ for $i=1,2,\dots,k$, and are also called *systematic sequences*. The remaining $n-k$ coded sequences are called *parity sequences*.

Similar to Theorem 2, we have the following result about the storage overhead of systematic shift-XOR codes:

Theorem 3. *The storage overhead of an $[n,k]$ shift-XOR code with* systematic *two-tone generator matrix $\mathbf{\Psi} = \begin{bmatrix} \mathbf{I} \\ \mathbf{\Phi} \end{bmatrix}$ is lower bounded by $\frac{((n-k)^2-1)(k-1)}{4}$ when $n-k$ is odd, and $\frac{(n-k)^2(k-1)}{4}$ when $n-k$ is even, and the lower bound is achieved if and only if $\mathbf{\Phi}$ is the reflected Vandermonde matrix.*

Proof. Similar to the proof of Theorem 2. Omitted. □

Algorithm 2 Decoding Algorithm for Systematic Shift-XOR Storage Codes

```
Input: bits stored in sequences $\hat{\mathbf{x}}_v$ $(1 \le v \le k - k_m)$, $h_1 < \cdots < h_{k-k_m}$.
Output: decoded message, stored in $\hat{\mathbf{x}}_v$ $(1 \le v \le k - k_m)$.
1: for $v \leftarrow 1 : (k - k_m)$ do
2:     for $u \leftarrow (k - k_m + 1) : k$ do
3:         for $\ell \leftarrow \max(0, t_{i_v,h_v} - t_{i_v,i_u}) + 1 : \min(0, t_{i_v,h_v} - t_{i_v,i_u}) + L$ do
4:             $\hat{\mathbf{x}}_v[\ell - t_{i_v,h_v} + t_{i_v,i_u}] \oplus \leftarrow \mathbf{x}_{i_u}[\ell]$
5: Apply Algorithm 1 on $\hat{\mathbf{x}}_v$, $v = 1, \ldots, k - k_m$.
```

Let us discuss the decoding algorithm of a storage system employing an $[n, k]$ systematic shift-XOR code. Assume that the decoder has access to k nodes with the indices in descending order $i_1 > i_2 > \cdots > i_k$. The k nodes have k_m systematic sequences and $k - k_m$ parity sequences. As the systematic sequences have smaller indices than the parity sequences, node i_u, $k - k_m + 1 \le u \le k$ stores the message sequence $\mathbf{x}_{i_u}$. Denote the indices of the remaining $k - k_m$ message sequence to decode as $1 \le h_1 < h_2 < \cdots < h_{k-k_m} \le k$.

The decoding scheme consists of two stages: the transmission stage and the decoding stage. In the transmission stage: first, the decoder retrieves $\mathbf{x}_{i_u}$ from node i_u for $k - k_m < u \le k$; second, the decoder retrieves $\hat{\mathbf{x}}_v = \mathbf{y}_{i_v}[t_{i_v,h_v} + (1 : L)]$ from node i_v for $1 \le v \le k - k_m$. In the decoding stage, $\mathbf{x}_{i_u}$, $k - k_m < u \le k$ are first substituted into $\hat{\mathbf{x}}_v$, $1 \le v \le k - k_m$. After the substitution, $\hat{\mathbf{x}}_v$, $1 \le v \le k - k_m$ form a two-tone system. Then the two-tone elimination is executed on $\hat{\mathbf{x}}_v$, $1 \le v \le k - k_m$ to decode $\mathbf{x}_{h_v}$ for $1 \le v \le k - k_m$. A pseudocode of the decoding stage is shown in Algorithm 2 to illustrate an in-place implementation. After the execution of Algorithm 2, $\hat{\mathbf{x}}_v$ becomes $\mathbf{x}_{h_v}$ for $1 \le v \le k - k_m$.

In Algorithm 2, the substitution (Line 1–4) costs no more than $(k - k_m)k_m L$ XOR operations and the two-tone elimination on $k - k_m$ sequences costs no more than $(k - k_m)(k - k_m - 1)L$ XOR operations. As a consequence, the number of XOR costs of Algorithm 2 is at most $(k - k_m)k_m L + (k - k_m)(k - k_m - 1)L = (k - k_m)(k - 1)L$. Similar to the two-tone elimination, Algorithm 2 costs $O(k^2 L)$ integer operations. As the substitution (Line 1–4) and two-tone elimination are in-place, Algorithm 2 is in-place implementable.

Systematic codes have the advantage of lower encoding/decoding complexity compared with the corresponding non-systematic codes, and also have lower storage overheads. The storage overheads of different shift-XOR codes, including non-systematic and systematic versions, with some typical parameters $[n, k]$ are shown in Table I. From the table, we see that the storage overheads of systematic codes are less than 10% of that of the corresponding non-systematic codes. Moreover, two-tones systematic codes have about $1/3$ storage overheads of that of previous systematic codes.

TABLE I
STORAGE OVERHEAD COMPARISON OF DIFFERENT SHIFT-XOR CODES. IN THE TABLE, RID CODES USE THE VANDERMONDE GENERATOR MATRICES, TWO-TONE CODES USE THE REFLECTED VANDERMONDE GENERATOR MATRICES, SYSTEMATIC INCREASING DIFFERENCE CODES USE THE VANDERMONDE GENERATOR MATRICES FOR PARITY SEQUENCES, AND SYSTEMATIC TWO-TONE CODES USE THE REFLECTED VANDERMONDE GENERATOR MATRICES.

Codes	$[8, 6]$	$[11, 8]$	$[14, 10]$
RID code [12], [15]	140	385	819
two-tone code	80	210	441
systematic incre. diff. code [14]	15	42	90
systematic two-tone code	5	14	36

IV. Implementation and Performance Analysis

In the previous two sections, we discussed the design of two-tone shift-XOR storage codes. In this section, we implement the two-tone shift-XOR codes and demonstrate the superior encoding and decoding throughputs compared with the state-of-the-art implementations of RS codes and Cauchy RS codes. In particular, we consider the following existing libraries for performance comparison:

- The Jerasure library [17] integrates several MDS codes, such as RS codes, Cauchy RS codes and RAID Liberation codes, and has been used in many academic projects.
- The Longhair library [18] is an implementation of Cauchy RS codes. Compared with the Jerasure library, this library improves the performance of Cauchy RS codes by introducing further optimizations, and performs 3 times faster than the Jerasure library.
- The Intel Intelligent Storage Acceleration Library (ISA-L) [19] is a collection of optimized low-level functions targeting storage applications, including RS codes, RAID Liberation codes. Written in assembly language, it highly optimizes the performance for Intel processors, and is the fastest implementation of RS codes to our knowledge.

A. Implementation Overview

We implement the systematic two-tone storage codes in C++. We first introduce the two basic routines for encoding and decoding.

1) Encoding Routine: The encoding routine of the systematic $[n, k]$ shift-XOR storage code takes k message sequences as input, and produces n coded sequences as output, among which k coded sequences are identical to the message sequences and the other $n - k$ sequences are parity sequences. Given encoding parameter $[n, k]$, the generator matrix is given in (9), where $\mathbf{\Phi}$ is a reflected Vandermonde matrix.

2) Decoding Routine: The decoding routine first determines whether the sequence is intact or corrupted. A bundle of sequences are decodable if the number of corrupted sequences is less or equal to $n - k$, otherwise it is undecodable. If the bundle is decodable, the decoding routine then recovers original message sequences using Algorithm 2, which first uses intact systematic sequences to perform substitution on intact parity sequences, and then uses Algorithm 1 to decode the remaining sequences.

TABLE II
STORAGE OVERHEAD OF TWO-TONE SHIFT-XOR CODES WITH DIFFERENT WORD SIZES w, WHERE THE UNIT OF THE OVERHEAD IS BYTE.

Codes	**[8, 6]**	**[11, 8]**	**[14, 10]**
Storage overhead for $w = 8$	5 B	14 B	36 B
Storage overhead for $w = 64$	40 B	112 B	288 B
Storage overhead for $w = 256$	160 B	488 B	1152 B

TABLE III
IMPLEMENTATION SPECIFICATIONS

	Hardware Support				
Name	SSE3	SSE4	AVX	AVX2	**Applied Method**
Jerasure	✓				Vandermonde RS
Longhair	✓				Cauchy RS
ISA	✓	✓	✓	✓	Vandermonde RS
Ours	✓	✓	✓	✓	Shift XOR

3) Implementation of Shift and XOR Operations: In section II and III, we described the encoding and decoding algorithms with respect to bit sequences, in which the unit of the operands in shift and XOR operations is one bit. However, in computer programs, the unit of operands is multiple bits, such as *char* (8 bits), *short* (16 bits), and *long* (64 bits), subject to the hardware supports. Suppose that the operands have the unit of w bits, called a *word*. We discuss how the shift and XOR operations used in shift-XOR codes are performed, and the effect on storage overheads.

A message sequence in our program has L words. We implement the shift and XOR operations as following:

- **Shift operation**: To implement the shift operation, we access the sequence elements by offsetting the indices. For the shift operator z^t, the offset should be t *words* instead of bits.
- **XOR operation**: The XOR operation performed on two words produces a word where each bit is the XOR of the corresponding bits in the two words.

To see why our previous discussion in term of bits can be transformed to words, we may regard that the word-wise shift/XOR operation is performing w bit-wise shifts/XORs in parallel. Therefore, encoding a shift-XOR code using sequences of L words can be regarded encoding w shift-XOR codes using sequences of L bits. As the word-wise shift/XOR operation performing the bit-wise operation in parallel, exploiting a larger word size could achieve higher computation efficiency, which would be discussed further in the next subsection.

However, exploiting a larger word size also increases the storage overhead: the effective storage overhead using a word of w bits is w times the storage overhead we have discussed in Section III. Table II gives the effective storage overheads of the $[8, 6]$, $[11, 8]$ and $[14, 11]$ shift-XOR codes with word sizes 8, 64 and 256 bits.

B. Optimization Techniques

We mainly apply two optimization techniques to achieve high performance in our implementation.

1) Vectorization: Modern CPU supports a type of data-level parallelism named SIMD (Single Instruction Multiple Data), such as SSE (Streaming SIMD Extensions) and AVX (Advanced Vector Extensions). In execution with SIMD, multiple operands can be operated simultaneously with one instruction. Our program heavily uses XOR in the calculation, so we utilize the AVX2 instruction set to accelerate the XOR calculation. Many existing libraries also exploit SIMD to accelerate the program, including the three libraries we selected for performance comparison. The detailed implementation specifications are shown in Table III.

In AVX2 SIMD programs, the size of the supported vector is 256 bits. Theoretically, programs that exploit AVX2 SIMD instructions would achieve 4 times acceleration in throughput compared with programs that do not use any SIMD instruction, because four 64-bit integers can be packed into one vector, and the XOR of two 256-bit vectors can be performed in one instruction. To align with the vector size in AVX2, the word size in our implementation is $w = 256$ bits.

Three intrinsic functions supported in AVX2 intrinsic set are used to calculate the XOR of two sequences and store the result into the destination sequence. In each execution of the XOR operation, the *_mm256_loadu_si256* function is first invoked twice to load two 256-bit words from memory into two SIMD registers, respectively, and then the *_mm256_xor_si256* function operates XOR on these two SIMD registers. After all of the execution finished, the *_mm256_storeu_si256* is invoked to store the content of the destination SIMD register into the memory of the destination sequence.

2) Cache Blocking: In modern computer systems, accessing data in the cache is faster than accessing the memory. A standard computer may have multiple levels of cache, such as L1, L2, and L3 cache. Processing units normally takes several cycles to access L1 cache, takes around 10 cycles to access L2 cache, takes tens of cycles to access L3 cache, and takes hundreds of cycles to access the memory. As the cache is also more expensive than memory, the size of cache space is much smaller, and hence it is not possible to store all the data in the cache during the encoding/decoding procedure. If for encoding each coded sequence, all the message sequences have to be loaded from memory, the time consumed by memory access would be more than the XOR operations and becomes the bottleneck of the overall throughput.

Since the processor preserves the recently used or most frequently used data in cache, it is commonly an optimization spotlight to better utilize the data already preserved in the faster cache space. If the processor cannot find a data reference in the cache space, a *cache miss* would happen so program need to fetch data from the lower memory hierarchy with latency penalty. A high *cache-miss rate* in execution indicates the program hardly reuses the data in cache and the program has to frequently fetch data from memory. Thereupon, we applied a technique named *cache blocking* [20] to reduce the cache-miss rate.

Algorithm 3 Cache Blocked Loop for Substitution in Algorithm 2.

```
1: $\ell_0 \leftarrow \max(0, t_{i_v,h_v} - t_{i_v,i_u}) + 1$
2: $\ell_{\max} \leftarrow \min(0, t_{i_v,h_v} - t_{i_v,i_u}) + L$
3: for $g \leftarrow 0 : \lceil(\ell_{\max} - \ell_0 + 1)/g_b\rceil - 1$ do
4:     for $u \leftarrow k - k_m + 1 : k$ do
5:         for $v \leftarrow 1 : k - k_m$ do
6:             for $\ell \leftarrow \ell_0 + g * g_b : \ell_0 + (g+1) * g_b - 1$ do
7:                 $\hat{\mathbf{x}}_v[\ell - t_{i_v,h_v} + t_{i_v,i_u}] \oplus \leftarrow \mathbf{x}_{i_u}[\ell]$
```

We focus on cache blocking optimization of a nested loop performing XOR operations as shown in Line 1–4 of Algorithm 2. The optimization applies to a similar loop used in the encoding routine as well. In the decoding routine this loop loads each word from the message sequences and performs substitution in the parity sequences. Essentially, this loop traversals each word in the systematic sequences, and performs an XOR operation with the corresponding word in the parity sequence. We perform two modifications to the loop in order to obtain a good data locality and reduce the cache-miss rate.

- Firstly, we swap the order of the out-most loop and the second out-most loop, such that multiple parity sequences can be generated from data already preserved in register.
- Secondly, we add a new loop that wraps the nested loop in order to block the data access. We introduce a parameter g_b in our program to control the cache block size and tune the block size to fit the size of L1 cache. The modified loop is shown as Algorithm 3.

Assume that the L1 cache has a size of c bytes. For encoding an $[n, k]$ code with word size w, setting g_b less than $\frac{8c}{wn}$ will have most data accessing blocked in the L1 cache. Considering the mechanism of cache line and instruction level parallelism, it is also not appropriate to set g_b too small. We conduct several experiments to obtain a suitable choice of g_b. In these experiments, we execute the program on different g_b, and collect the L1 cache-miss rate[1] and instructions per cycle[2] [21]. The result is shown in Table IV. From this result we can see that programs that exploit cache blocking have lower cache-miss rates and higher instructions per cycle compared with the program that does not exploit any cache blocking. Guided by this experiment, we set g_b in our program to 16.

TABLE IV
PERFORMANCE METRICS FOR DIFFERENT g_b. HERE THE WORD SIZE IS $w = 256$ BITS AND THE NUMBER OF CODED SEQUENCES IS $n = 11$.

g_b	L1 Cache Miss Rate	Instruction Per Cycle
4	4.94%	1.9
8	5.43%	2.37
16	5.04%	2.42
32	6.06%	2.29
64	8.15%	2.35
No Blocking	23.74%	0.77

C. Experiment Setup and Results

For storage codes employed by commercial distributed storage systems, such as Google File System, Facebook Hadoop Distributed File System and Windows Azure Storage, the size of coded sequences is commonly set to 1.33 to 2 times of the size of original message sequences. Hence, we choose 3 sets of $[n, k]$ parameters to meet the settings in the real scenarios: $[8, 6]$, $[11, 8]$ and $[14, 10]$.

[1]L1 cache miss rate counts the ratio of data references that cannot be found on L1 cache.

[2]Instructions per cycle is the average number of instructions executed for each cycle. It is an important metric to indicate how efficient the program utilizes the computer micro-architecture.

Our experiment evaluates the performance on different file sizes, which can show the effect of different levels of cache and memory on the throughputs. Specifically, we choose 7 different file sizes from 128KB to 512MB.

Let t_{enc} and t_{dec} denote the time of encoding $n - k$ parity sequences and decoding k original message sequences, respectively. Then the throughputs of encoding and decoding process are measured as $\frac{nLw}{t_{\text{end}}}$ and $\frac{kLw}{t_{\text{dec}}}$, where the unit is bit per second.

We compile our implementation and other libraries for comparison by g++ with O2 level optimization and create a dummy file with randomized content. The testing environment is a dedicated server with Intel Xeon CPU E5-2699 v4 at 2.2GHz. The memory size is 378GB, the L1 cache size is 32KB, the L2 cache size is 256KB, and the L3 cache size is 56320KB. Here, the time of loading message sequences into memory and the time of writing coded sequences to the disk are not involved in the calculation of encoding time. And the time of loading coded sequences into memory and the time of writing message sequences to the disk are not involved in the calculation of decoding time. Table V and Table VI show the encoding throughputs and decoding throughputs on 7 different file sizes among the 4 implementations. All of the data in the table is the average over 10000 runs.

From the tables, we can observe that the shift-XOR code implementation outperforms all the other coding libraries in both encoding and decoding throughputs. The shift-XOR code implementation achieves from 40 to 80 percent more throughputs than the state-of-the-art coding library ISA-L for both encoding and decoding performance. Compared with Cauchy-RS codes implemented in the Longhair library, our codes can achieve 80% higher encoding/decoding throughputs for small file size (128KB) and from 5 to 8 times the encoding/decoding throughputs for large file size (512MB). Compared with the RS codes implemented in Jerasure library, our codes can achieve 10 times the encoding/decoding throughput.

We observe a significant throughput drop occurs when the file size changes from 32MB to 128MB. This is because the size of cache in the experiment system is around 60MB. For encoding/decoding with input file size larger than the size of cache, it is not feasible to access all the data in the cache, and longer time is needed to access data in memory, which results in a significant drop in throughputs.

TABLE V
ENCODING THROUGHPUT IN MEGABYTES PER SECOND.

(a) $n = 8, k = 6$

File Size	Two-tone	ISA-L	Jerasure	Longhair
128 KB	16,548	10,096	1,594	9,605
512 KB	16,854	11,934	1,618	7,960
1 MB	16,793	9,760	1,154	6,321
32 MB	16,843	9,054	820	4,899
128 MB	12,098	6,518	802	2,837
256 MB	10,155	6,419	803	2,209
512 MB	10,110	6,424	808	2,124

(b) $n = 11, k = 8$

File Size	Two-tone	ISA-L	Jerasure	Longhair
128 KB	11,823	7,381	1,078	4,106
512 KB	12,256	7,628	1,272	4,097
1 MB	12,702	6,843	1,154	3,613
32 MB	12,086	6,436	773	2,283
128 MB	7,952	5,177	659	1,455
256 MB	7,926	5,059	660	1,193
512 MB	7,888	4,992	592	1,110

(c) $n = 14, k = 10$

File Size	Two-tone	ISA-L	Jerasure	Longhair
128 KB	8,725	5,244	860	2,769
512 KB	10,521	6,303	869	2,608
1 MB	9,145	6,706	791	2,233
32 MB	7,859	5,003	516	1,337
128 MB	6,227	4,057	453	937
256 MB	5,647	3,879	453	828
512 MB	5,450	3,801	449	744

TABLE VI
DECODING THROUGHPUT IN MEGABYTES PER SECOND.

(a) $n = 8, k = 6$

File Size	Two-tone	ISA-L	Jerasure	Longhair
128 KB	11,936	8,036	1,204	9,145
512 KB	11,125	8,262	1,155	6,319
1 MB	10,213	6,803	895	4,410
32 MB	6,161	4,590	638	1,390
128 MB	6,227	3,635	581	1,139
256 MB	6,033	3,327	585	971
512 MB	6,252	3,140	556	933

(b) $n = 11, k = 8$

File Size	Two-tone	ISA-L	Jerasure	Longhair
128 KB	8,021	5,205	783	3,368
512 KB	7,442	5,773	837	2,953
1 MB	7,474	4,762	840	2,519
32 MB	7,145	3,477	564	1,149
128 MB	4,542	2,919	437	827
256 MB	4,531	2,730	476	710
512 MB	4,522	2,556	387	650

(c) $n = 14, k = 10$

File Size	Two-tone	ISA-L	Jerasure	Longhair
128 KB	4,883	3,013	604	2,677
512 KB	4,713	3,704	562	2,488
1 MB	5,468	3,559	511	2,120
32 MB	4,933	2,962	360	1,286
128 MB	3,273	2,686	337	627
256 MB	3,019	2,490	314	543
512 MB	3,156	2,380	300	496

V. Conclusion

In this paper, we proposed a new class of shift-XOR storage codes with smaller storage overhead by using two-tone generator matrices, which generalize the previous (refined) increasing difference matrices. We extended the shift-XOR elimination to handle two-tone shift-XOR codes. Towards practical applications, we discussed the storage code design with reflected Vandermonde matrices and systematic shift-XOR codes. We implemented the systematic two-tone shift-XOR storage codes using C++, which demonstrated $40\% \sim 80\%$ higher encoding/decoding throughput than the state-of-the-art encoding/decoding library used by industry.

Appendix

Lemma 1. *For a two-tone generator matrix $\mathbf{\Psi} = (z^{t_{i,j}})$ and $1 \le j < j' \le k$ we have,*

$$\begin{cases} t_{i,j} - t_{i,j'} \ge (d-i)(j'-j), & 1 \le i \le d, \\ t_{i,j'} - t_{i,j} \ge (i-d)(j'-j), & d < i \le n. \end{cases}$$

where the equalities hold if and only if

$$\begin{cases} t_{i,a} - t_{i,a+1} = d - i, & 1 \le i \le d, \\ t_{i,a+1} - t_{i,a} = i - d, & d < i \le n. \end{cases}$$

Proof. By the definition of two-tone property, we have for $1 \le b < b' \le k$

$$\begin{cases} 0 \le t_{a',b} - t_{a',b'} < t_{a,b} - t_{a,b'}, & 1 \le a < a' \le d, \\ 0 < t_{a,b'} - t_{a,b} < t_{a',b'} - t_{a',b}, & d < a < a' \le n. \end{cases}$$

(i) For $1 \le a < a' \le d$, let $a = i$, $a' = i+1$, $b' = j+1$, $b = j$, we have $t_{i,j} - t_{i,j+1} \ge t_{i+1,j} - t_{i+1,j+1} + 1$. Repeating the above process, we have

$$t_{i,j} - t_{i,j+1} \ge t_{d,j} - t_{d,j+1} + d - i.$$

Since $t_{d,j} - t_{d,j+1} \ge 0$, we have

$$t_{i,j+1} - t_{i,j} \ge d - i. \tag{10}$$

Similarly, we can derive inequalities

$$t_{i,a} - t_{i,a+1} \ge d - i, \quad \forall a = j+1, \dots, j'-1. \tag{11}$$

Summing up the inequalities in (10) and (11), we have $t_{i,j} - t_{i,j'} \ge (d-i)(j'-j)$, $1 \le i \le d$. It is noted that the equality holds if and only if all the equalities in in (10) and (11) hold.

(ii) For $d < a < a' \le n$, let $a' = i$, $a = i-1$, $b' = j+1$, $b = j$, we have $t_{i,j+1} - t_{i,j} \ge t_{i-1,j+1} - t_{i-1,j} + 1$. Repeating the above process, we have

$$t_{i,j+1} - t_{i,j} \ge t_{d+1,j+1} - t_{d+1,j} + i - d - 1.$$

Since $t_{d+1,j+1} - t_{d+1,j} > 0$, i.e., $t_{d+1,j+1} - t_{d+1,j} \ge 1$, we have

$$t_{i,j+1} - t_{i,j} \ge i - d. \tag{12}$$

Similarly, we can derive inequalities

$$t_{i,a+1} - t_{i,a} \ge i - d, \quad \forall a = j+1, \dots, j'-1. \tag{13}$$

Summing up the inequalities in (12) and (13), we have $t_{i,j'} - t_{i,j} \ge (i-d)(j'-j)$ for $d < i \le n$. Similarly, the equality holds if and only if all the equalities in (12) and (13) hold. $\square$

References

[1] I. Reed and G. Solomon, "Polynomial codes over certain finite fields," *Journal of the Society for Industrial and Applied Mathematics*, vol. 8, pp. 300–304, Jun. 1960.

[2] D. Borthakur, R. Schmidt, R. Vadali, S. Chen, and P. Kling, "HDFS RAID," in *Hadoop User Group Meeting*, 2010.

[3] A. Fikes, "Storage architecture and challenges," *Talk at the Google Faculty Summit*, vol. 535, 2010.

[4] D. Ford, F. Labelle, F. I. Popovici, M. Stokely, V. Truong, L. Barroso, C. Grimes, and S. Quinlan, "Availability in globally distributed storage systems," in *9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010, October 4-6, 2010, Vancouver, BC, Canada, Proceedings*, R. H. Arpaci-Dusseau and B. Chen, Eds. USENIX Association, 2010, pp. 61–74.

[5] J. Bloemer, M. Kalfane, R. Karp, M. Karpinski, M. Luby, and D. Zuckerman, "An XOR-based erasure-resilient coding scheme," 1995.

[6] J. S. Plank and L. Xu, "Optimizing Cauchy Reed-Solomon codes for fault-tolerant network storage applications," in *Fifth IEEE International Symposium on Network Computing and Applications (NCA'06)*. IEEE, 2006, pp. 173–180.

[7] M. Blaum and R. M. Roth, "New array codes for multiple phased burst correction," *IEEE Trans. Information Theory*, vol. 39, no. 1, pp. 66–77, 1993.

[8] M. Xiao, M. Médard, and T. Aulin, "A binary coding approach for combination networks and general erasure networks," in *IEEE International Symposium on Information Theory, ISIT 2007, Nice, France, June 24-29, 2007*. IEEE, 2007, pp. 786–790.

[9] M. Xiao, T. Aulin, and M. Médard, "Systematic binary deterministic rateless codes," in *2008 IEEE International Symposium on Information Theory, ISIT 2008, Toronto, ON, Canada, July 6-11, 2008*, F. R. Kschischang and E. Yang, Eds. IEEE, 2008, pp. 2066–2070.

[10] K. W. Shum and H. Hou, "Network coding based on byte-wise circular shift and integer addition," *CoRR*, vol. abs/2005.07336, 2020.

[11] H. Hou, K. Shum, M. Chen, and H. Li, "BASIC regenerating code: Binary addition and shift for exact repair," in *IEEE Int. Symp. Inf. Theory*, Jul. 2013, pp. 1621–1625.

[12] X. Fu, S. Yang, and Z. Xiao, "Recovery and repair schemes for shift-xor regenerating codes," *CoRR*, vol. abs/1907.05058, 2019.

[13] C. Sung and X. Gong, "A ZigZag-decodable code with the MDS property for distributed storage systems," in *IEEE Int. Symp. Inf. Theory*, Jul. 2013, pp. 341–345.

[14] X. Fu, Z. Xiao, and S. Yang, "Overhead-free in-place recovery scheme for XOR-based storage codes," in *IEEE Int. Conf. Trust, Security and Privacy in Computing and Communications*, Sep. 2014, pp. 552–557.

[15] ——, "Overhead-free in-place recovery and repair schemes of XOR-based regenerating codes," in *IEEE Int. Symp. Inf. Theory*, Jul. 2015, pp. 341–345.

[16] X. Gong and C. W. Sung, "Zigzag decodable codes: Linear-time erasure codes with applications to data storage," *J. Comput. Syst. Sci.*, vol. 89, pp. 190–208, 2017.

[17] J. S. Plank, S. Simmerman, and C. D. Schuman, "Jerasure: A library in c/c++ facilitating erasure coding for storage applications-version 1.2," *University of Tennessee, Tech. Rep. CS-08-627*, vol. 23, 2008.

[18] C. A. Taylor, "Longhair: Fast Cauchy Reed-Solomon Erasure Codes in C," 2018. [Online]. Available: https://github.com/catid/longhair

[19] Intel, "Intel(R) Intelligent Storage Acceleration Library," 2020. [Online]. Available: https://github.com/intel/isa-l

[20] M. D. Lam, E. E. Rothberg, and M. E. Wolf, "The cache performance and optimizations of blocked algorithms," in *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS IV. New York, NY, USA: Association for Computing Machinery, 1991, p. 63–74.

[21] Intel, "Intel(R) 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture." [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-1-manual.pdf