Fast Erasure Coding for Data Storage: A Comprehensive Study of the Acceleration Techniques

Tianli Zhou and Chao Tian, ***Texas A&M University***

https://www.usenix.org/conference/fast19/presentation/zhou

This paper is included in the Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST '19).

February 25–28, 2019 • Boston, MA, USA

978-1-939133-09-0

Open access to the Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST '19) is sponsored by

NetApp®

Fast Erasure Coding for Data Storage: A Comprehensive Study of the Acceleration Techniques

Tianli Zhou and Chao Tian
Texas A&M University

Abstract

Various techniques have been proposed in the literature to improve erasure code computation efficiency, including optimizing bitmatrix design, optimizing computation schedule, common XOR operation reduction, caching management techniques, and vectorization techniques. These techniques were largely proposed individually previously, and in this work, we seek to use them jointly. In order to accomplish this task, these techniques need to be thoroughly evaluated individually, and their relation better understood. Building on extensive test results, we develop methods to systematically optimize the computation chain together with the underlying bitmatrix. This led to a simple design approach of optimizing the bitmatrix by minimizing a weighted cost function, and also a straightforward erasure coding procedure: use the given bitmatrix to produce the computation schedule, which utilizes both the XOR reduction and caching management techniques, and apply XOR level vectorization. This procedure can provide a better performance than most existing techniques, and even compete against well-known codes such as EVENODD, RDP, and STAR codes. Moreover, the result suggests that vectorizing the XOR operation is a better choice than directly vectorizing finite field operations, not only because of the better encoding throughput, but also its minimal migration efforts onto newer CPUs.

1 Introduction

A leading technique to achieve strong fault-tolerance in data storage systems is to utilize erasure codes. Erasure codes have been widely used in various data storage systems, ranging from disk array systems [5], peer-to-peer storage systems [22], to distributed storage systems [9, 11], and cloud storage systems [4]. The root of erasure codes can be traced back to the well-known Reed-Solomon codes [20], or more generally, maximum distance separable codes [10]. Roughly speaking, such erasure codes allow a fixed number of component failures in the overall system, and it has the lowest storage overhead (i.e., redundancy) among all strategies that can tolerate the same number of failures. One example is Quantcast File System (QFS) [13], which is an implementation of the data storage backend for the MapReduce framework; it can save 50% of storage space over the original HDFS which uses 3-replication, while maintaining the same failure-tolerance capability.

It has long been recognized that encoding data into its erasure-coded form will incur a much heavier computation load than simple data replication [21], thus more time-consuming. In order to complete the coding computation more efficiently, various techniques have been proposed in the literature to either directly reduce this computation load [2, 5–8, 18], or to accelerate the computation by better utilizing the resources in modern CPUs [12, 16].

Erasure codes rely on finite field operations, and in computer systems, the fields are usually chosen to be $GF(2^w)$, that is, an extension field of the binary field. Using the fact that such finite field operations can be effectively performed using binary XOR between the underlying binary vectors and matrices [3], Plank et al. [18] proposed efficient methods to encode using the "bitmatrix" representation. Several techniques were introduced in the same work to reduce the number of the XOR operations in the computation, and the overall encoding procedure can be viewed as a sequence of such XOR operations, i.e., organized in a computation schedule. Huang et al. [7] (see also the Liberation codes [14] where a similar idea was mentioned) made the observation that some chains of XORs to compute different parity bits may have common parts, and thus by computing the common parts first, the overall computation can be reduced. A matching strategy was proposed to identify such common parts, which leads to more efficient computation schedules. Further heuristic methods to reduce the number of XOR's along these lines were investigated by Plank et al. [17], and lower bounds on the total number of XOR's have also been found [15].

Though with the same goal of reducing the computation load in mind, the coding theory community addresses the is-

sue from another perspective, where specific code constructions have been proposed. Several notable examples of such codes can be found in [2, 5, 6, 8]. These codes usually allow only two or three parities, instead of the flexible choices seen in generic erasure codes.

In contrast to the approaches discussed above where the computation load can be fundamentally reduced, a different approach to improve the encoding speed is to better utilize the existing computation resources in modern computers, i.e., hardware acceleration. Particularly, since modern CPUs are typically built with the capability of "single-instruction-multiple-data" (SIMD), sometimes referred to as vectorization, it was proposed that instead of using the bitmatrix implementation, erasure coding can be efficiently performed by vectorizing finite field operations directly [16]. It was shown that this approach can provide significant improvements over the approach based on the afore-mentioned bitmatrix representation without vectorization. Also related to this approach of optimizing resource utilization, Luo et al. [12] noted that the order of operations in the computation schedule of the bitmatrix-based approach can affect the performance, due to CPU cache miss penalty, and thus steps can be taken to optimize the cache access efficiency.

Although these existing works have improved the coding efficiency of erasure codes to more acceptable levels, the sheer amount of data in modern data storage systems implies that even a small improvement of the coding efficiency may provide significant cost saving and be an important performance differentiator. Particularly, virtualization has been widely adopted for cloud computing, and erasure coding on such cloud platform will be more resource-constrained than on the native platform, thus reducing the computation load is very meaningful. Against this general backdrop, in this work we seek to answer the following questions:

1. Which methods are the most effective, i.e., can provide the most significant improvement? Particularly, how to make a fair comparison of the two distinct approaches of optimizing bitmatrix schedules and vectorization?
2. Can and should these techniques be utilized together, in order to maximize the encoding throughput?
3. If these techniques can be utilized together, which component should be optimized and how to optimize them?

In the process of answering these question, we discovered a particularly effective approach to accelerate erasure encoding: selecting bitmatrices optimized for the weight sum of the number of XOR and copying operations, taking into consideration of the reduction from the common XOR chains, then using XOR-level vectorization for hardware acceleration. The resulting encoding process we propose can provide significant improved encoding throughput compared to [12, 16, 18], ranging from 20% to 500% for various parameters. Moreover, in most cases, the proposed approach can compete with the well-known EVENODD code [2], RAID-6 code [14], RDP code [6], STAR code [8], and triple-parity Reed-Solomon code in Quantcast-QFS [13], which were specifically designed for fast encoding and only for restricted parameters.

Our result also suggests that instead of vectorizing the finite field operation directly, we should vectorize the XOR operations based on the bitmatrix representation. In addition to the throughput advantage, this approach in fact has an important practical advantage: vectorizing general finite field operation involves software implementation of a larger set of relevant operations using the CPU-specific instructions (for different finite field sizes and different finite field operations), while vectorizing XOR operations essentially involves only a single such instruction. As newer versions of CPUs and instruction sets are introduced, the proposed approach only requires minimal migration effort, since most of the bitmatrix implementation is hardware agnostic.

2 Background and Review

2.1 Erasure Codes and Reed-Solomon Codes

Erasure codes are usually specified by two parameters: the number of data symbols k to be encoded, and the number of coded symbols n to be produced. The data symbols and the coded symbols are usually assumed to be in finite field $GF(2^w)$ in computer systems. Such erasure codes are usually referred to as the (n,k) erasure codes.

To be more precise, let k linearly independent vectors $g_0, g_1, \ldots, g_{k-1}$ (of length n each) be given, whose components are in the finite field $GF(2^w)$. Denote the data (sometimes referred to as the message) as $u = (u_0, u_1, \ldots, u_{k-1})$, whose components are also represented as finite field elements in $GF(2^w)$. The codeword for the message u is then

$$v = u_0 g_0 + u_1 g_1 + \cdots + u_{k-1} g_{k-1}.$$

This encoding process can alternatively be represented using the *generator matrix* G of dimension $k \times n$ as

$$v = u \cdot G, \tag{1}$$

where

$$G = \begin{bmatrix} g_0 \\ g_1 \\ \vdots \\ g_{k-1} \end{bmatrix} = \begin{bmatrix} g_{0,0} & g_{0,1} & \cdots & g_{0,n-1} \\ g_{1,0} & g_{1,1} & \cdots & g_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ g_{k-1,0} & g_{k-1,1} & \cdots & g_{k-1,n-1} \end{bmatrix}.$$

In most data storage applications, the erasure codes have the maxmium distance separable (MDS) property, meaning that the data can be recovered from any k coded symbols in the vector v. In other words, it can tolerate loss of any $m = n - k$ symbols. This property can be guaranteed, as long as any k-by-k submatrix of G, which is created by deleting any m columns from G, is invertible.

2.1.1 Reed-Solomon Code

The original Reed-Solomon code relies on a Vandermonde matrix to guarantee that this invertibility condition is satisfied, i.e.,

$$G = \begin{bmatrix} 1 & \cdots & 1 & \cdots & 1 \\ a_0 & \cdots & a_i & \cdots & a_{n-1} \\ a_0^2 & \cdots & a_i^2 & \cdots & a_{n-1}^2 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_0^{k-1} & \cdots & a_i^{k-1} & \cdots & a_{n-1}^{k-1} \end{bmatrix} \quad (2)$$

where a_i's are distinct symbols in $GF(2^w)$.

Using a generator matrix of the Vandermonde form will produce a non-systematic form of the message, i.e., the message u is not an explicit part of the codeword v. We can convert G through elementary row operations (see e.g., [10]) to obtain an equivalent generator matrix G'

$$G' = [I, P] = \begin{bmatrix} 1 & 0 & \cdots & 0 & p_{0,0} & \cdots & p_{0,m-1} \\ 0 & 1 & \cdots & 0 & p_{1,0} & \cdots & p_{1,m-1} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & 1 & p_{k-1,0} & \cdots & p_{k-1,m-1} \end{bmatrix}$$

where the left portion is the identity matrix I_k of dimension k-by-k, and the right portion is the "parity coding matrix" P. As a consequence, we have

$$v = u \cdot G' = (u_0, u_1, \cdots, u_{k-1}, p_0, p_1, \cdots, p_{m-1}), \quad (3)$$

where

$$(p_0, p_1, \cdots, p_{m-1}) = (u_0, u_1, \cdots, u_{k-1}) \cdot P. \quad (4)$$

The matrix P is sometimes also referred to as the coding distribution matrix [19].

2.1.2 Cauchy Reed-Solomon Codes

Instead of reducing from a Vandermonde generator matrix, we can also directly assign the matrix P such that the invertible condition can be satisfied. One well-known choice is to let P be a Cauchy matrix, and the corresponding erasure code is often referred to as Cauchy Reed-Solomon (GRS) codes [3].

More precisely, denote $X = (x_1, \ldots, x_k)$ and $Y = (y_1, \ldots, y_m)$, where x_i's and y_i's are distinct elements of $GF(2^w)$. Then the element in row-i column-j in the Cauchy matrix is $1/(x_i + y_j)$. It is clear that any submatrix of a Cauchy matrix is still a Cauchy matrix. Particularly, let C_ℓ be an order-ℓ square submatrix of a Cauchy matrix:

$$C_n = \begin{bmatrix} \frac{1}{x_1+y_1} & \frac{1}{x_1+y_2} & \cdots & \frac{1}{x_1+y_\ell} \\ \frac{1}{x_2+y_1} & \frac{1}{x_2+y_2} & \cdots & \frac{1}{x_2+y_\ell} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{1}{x_\ell+y_1} & \frac{1}{x_\ell+y_2} & \cdots & \frac{1}{x_n+y_\ell} \end{bmatrix},$$

then C_ℓ is invertible, and the elements of the inverse of the Cauchy matrix C_ℓ^{-1} have an explicit analytical form [3].

One advantage of using Cauchy Reed-Solomon code instead of the classical Reed-Solomon code based on Vandermonde matrix is that inverting an order-n Vandermonde-based matrix is of time complexity $O(n^3)$, while inverting a Cauchy matrix has a time complexity $O(n^2)$. Following [18], we adopt Cauchy Reed-Solomon codes in this work, instead of the Vandermonde matrix based approach.

2.2 Encoding by Bitmatrix Presentation

Finite field operations in $GF(2^w)$ can be implemented using the underlying bit vectors and matrices [3], and thus all the computations can be conducted using direct copy or binary XOR. Based on this representation, reducing erasure code computation is equivalent to reducing the number of XOR and copying in the computation schedule. Various techniques to optimize this metric have been proposed in the literature, which we also briefly review in this subsection.

2.2.1 Convert Parity Matrix to Bitmatrix

Each element e in $GF(2^w)$ can be represented as a row vector $V(e)$ of $1 \times w$ or a matrix $M(e)$ of $w \times w$, where each element in the new representation are in $GF(2)$. $V(e)$ will be identical to the binary representation of e, and the i^{th} row in $M(e)$ is $V(e^{2^{i-1}})$. If we apply this representation, the parity coding matrix of size $k \times m$ will be converted to a new parity coding matrix of size $wk \times wm$ in $GF(2)$, i.e., a binary matrix. Using the bitmatrix representation, erasure coding can be accomplished by XOR operations, together with an initial copying operation. A simple example of bitmatrix encoding is shown in Figure 1, where the matrix multiplications are now converted to XORs of data bits corresponding to the ones in the binary parity coding matrix, together with some copying operations.

The number of 1's in the bitmatrix is the number of XOR operations in encoding. Choosing different $X = (x_1, \ldots, x_k)$ and $Y = (y_1, \ldots, y_m)$ vectors will produce different encoding bitmarices, which have different numbers of 1's and thus different numbers of XOR operations in the computation schedule. In [19], exhaustive search and several other heuristics were used to find better assignments of the (X, Y) vector such that the number of 1's in the bitmatrix can be reduced. It was shown that these techniques can lead to encoding throughput improvement ranging from 10%-17% for different (n, k, w) parameters.

2.2.2 Normalization of the Parity Coding Matrix

A simple procedure to reduce the encoding computations is to multiply each row and each column of $G = [I, P]$ by certain non-zero values, such that some of the coefficients are more

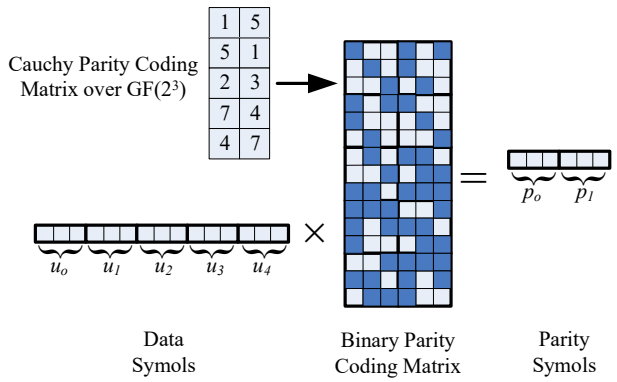

Figure 1: Encoding in bitmatrix representation for $k = 5, m = 2, w = 3$: the parity coding matrix is first converted to its bitmatrix form (blue as bit 1, and gray as 0), and the encoding is done as a multiplication of the length-15 binary vector and the 15×6 binary matrix.

suitable for computation (e.g., to make some elements be the multiplicative unit 1), which is referred to as bitmatrix normalization [18]. Clearly this does not change the invertibility property of the original generator matrix. More precisely, for any parity coding matrix P, we can use the following procedure:

1. For each row-i in P, divide each element by $p_{i,0}$, after which all elements in column-0 will be the multiplicative unit in the finite field.

2. For each column-j except the first:

   (a) Count the number of ones in the column (in the bitmatrix representation of this column).

   (b) Divide column-j by $p_{i,j}$ for each i, and count the number of ones in this column (in the bitmatrix representation).

   (c) Using the $p_{i,j}$ which yields the minimum number of ones from the previous two steps, let the new column-j be the values after the division of $p_{i,j}$. In other words, we normalize column-j with the element in the column which induces the minimum number of ones in the bitmatrix.

An example given by Plank et al. [18] shows that for $m = 3$, this procedure can reduce the number of ones in the bitmatrix to 34 ones from the original 46. This method is rather straightforward to implement and does not require any additional optimization.

2.2.3 Smart Scheduling

The idea of reusing some parity computation to reduce overall computation can be found in the code construction proposed by Plank [14], and this idea materialized as the smart scheduling component in the software package [18]. The underlying idea is as follows: if a parity bit can be written as the XOR of another parity bit and a small number of data bits, then it can be computed more efficiently. The following example should make this idea clear. Suppose the two parities are given as

$$p_0 = u_0 \oplus u_2 \oplus u_3 \oplus u_4, \quad p_1 = u_0 \oplus u_2 \oplus u_3 \oplus u_5. \tag{5}$$

A direct implementation to generate p_1 will use 3 XOR operations, but by utilizing p_0, p_1 can be computed as

$$p_1 = p_0 \oplus u_4 \oplus u_5,$$

which requires only two XORs. This technique requires slightly more effort to implement and optimize than the previous technique, however the computation schedule can essentially be generated offline and thus it does not contribute significantly to the encoding computation.

2.2.4 Matching

The idea of smart scheduling in fact has a related form. Huang et al. [7] recognized that instead of restricting to reusing computed parity bits, any common parts of the XOR chains in computing the parity bits can be reused, which reduces the total number of XOR computations. A grouping strategy was consequently proposed to optimize the number of necessary XORs. The proposed method focuses only on common XOR operations involving a pair of data bits, but not common operations involving three or more data bits. This is because common operations of three or more data bits are scarce in practical codes, and at the same time identifying them can be time consuming.

The core idea of the proposed approach by Huang et al. (common operation first) is to represent the demand data pair of parities in a graph, each vertex of which corresponds to an input data bit, and the weight of edge between vertex i and j represents the number of parities demands $u_i \oplus u_j$. A greedy procedure is used to extract a sub-graph with the edges of the largest weights, then the *maximum cardinality matching* algorithm can be used on the sub-graph to find a set of edges, where none of them have shared vertices. Each edge such found indicates a pair of input data bits whose XOR is common in some XOR chains. The algorithm then removes these edges and vertices from the original graph, and repeat this subgraph extraction and matching procedure on the remaining graph. This technique requires further effort to implement and optimize than smart scheduling, but the computation schedule can also be generated offline.

In the matching phase on the sub-graphs, two different strategies were introduced:

1. Unweighted matching. This method views all edges in the graph as having the same weight.

2. Weighted matching. This method uses the heuristic of making the matching covers as few dense nodes in the sub-graph (defined as degrees of nodes) as possible, by adjusting the assignments of the weights on the edges according to the sum of degrees of both ends in the original graph.

In our work, an implementation of Edmond's blossom shrinking algorithm in the LEMON graph library [1] is utilized to implement these matching algorithms.

Generalizing the matching technique, a few more heuristic methods to reduce the number of XOR's were investigated by Plank et al. [17]. However, these methods themselves can be extremely time-consuming, and the observed improvements in encoding throughput appear marginal. Therefore, we do not pursue these heuristic methods in this work.

2.3 Optimizing Utilization of CPU Resources

Speeding-up erasure coding computation can also be obtained through more efficient utilization of CPU resources, such as vectorization and avoiding cache misses.

2.3.1 Vectorization for Hardware Acceleration

Modern CPUs typically have SIMD capability, which can dramatically improve the computation speed. Most of the computation in erasure coding can be done in parallel, including XOR operations and more general finite field operations, since the same operations need to be applied on an array of data.

Directly vectorizing finite field operations has been previously investigated and implemented for finite fields $GF(2^8)$, $GF(2^{16})$, and $GF(2^{32})$ [16]. This was accomplished through invoking the 128-bit SSE vectorization instruction set for INTEL and AMD CPUs. More recently, 256-bit AVX2 vectorization instructions and 512-bit AVX-512 vectorization instructions are becoming more common in newer generations of CPUs. In this work, we only report results based on the 128-bit SSE instruction set in order to make the comparison fair, however, our implementation can indeed utilize 256-bit AVX2 vectorization instructions and 512-bit AVX-512 vectorization instructions without any essential change to the software program, exactly because of the reason mentioned at the end of Section 1.

2.3.2 Reducing Cache Misses

The sequence of operations can affect the coding performance due to cache misses, and more efficient cache-in and cache-out can be accomplished by choosing a streamlined computation order. A detailed analysis of the CPU-cache handling and the effects of different operation orders was given by Luo et al. [12]. The conclusion is that increased spatial data locality can help to reduce cache miss penalty. Consequently, a computation schedule was proposed where a data chunk is accessed only once sequentially, each of which is then used to update all related parities. More precisely, this strategy will read the data symbol u_0 first, update all parities which involves u_0, then $u_1, u_2, ..., u_{k-1}$. In contrast, the naive strategy of computing the parity symbols $p_0, p_1, ..., p_{m-1}$ sequentially suffers a performance loss, which was reported to be roughly 23%∼36%.

3 Effects of Individual Techniques and Possible Combinations

As mentioned earlier, the first step of our work is to better understand the effects of the existing techniques in speeding up the erasure coding computation. For this purpose, we first conduct tests on encoding procedures with each individual component enabled. The relation of different techniques will be discussed later, which allows us to utilize them together in the proposed design procedure.

3.1 Analyzing Individual Techniques

The existing techniques we consider individually are: XOR bitmatrix normalization (BN), XOR operation smart scheduling (SS), common XOR reduction using unweighted matching (UM), common XOR reduction using weighted matching (WM), scheduling for caching optimization (S-CO), and direct vectorization of XOR operation (V-XOR). The first four techniques can be viewed as optimization on the bitmatrix such that the total number of XOR (and copy) operations is reduced, as discussed in Section 2.2. The last technique, though has not been systematically investigated in the literature, is in fact a rather natural choice and is thus included in our test. The latter two methods aim to better utilize the CPU resources such that the computation can be done more efficiently without reducing fundamentally the computation load, and they are more hardware platform dependent.

We conducted encoding throughput tests for a range of (n,k,w) parameters most relevant for data storage applications, the results of which are reported in Table 1, in terms of the improvement over the baseline approach of taking a simple Cauchy Reed-Solomon code without any additional optimization. For the first four techniques, the improvement is measured in terms of the reduction of the number of 1's in the bitmatrix, while for the latter two, the improvement is measured in terms of the encoding throughput increase. Multiple tests are performed for each parameter, and we report the average over them. In the last row of Table 1, the average encoding throughput over all the tested parameters is included. All tests in this work are conducted on a workstation with an AMD Ryzen 1700X CPU (8 cores) running at 3.4GHz, 16GB DDR4 memory, which runs the Ubuntu 18.04 64-bit operating system and the compiler is GCC 7.3.0

Table 1: Performance improvements by individual techniques

(n,k,w)	The number of XOR's in the baseline code	Reduction in the number of XOR's				Throughput increase	
		BN	SS	UM	WM	S-CO	V-XOR
(8,6,4)	104	42.31%	17.31%	31.73%	31.73%	2.14%	98.44%
(8,6,8)	362	53.31%	33.70%	34.53%	34.81%	0.52%	126.36%
(9,6,4)	152	32.89%	19.74%	32.89%	32.89%	3.54%	90.48%
(9,6,8)	549	44.63%	29.14%	38.25%	38.25%	4.51%	152.87%
(10,6,4)	200	27.50%	22.00%	36.00%	36.00%	0.60%	91.30%
(10,6,8)	736	40.90%	28.80%	38.59%	38.59%	1.23%	130.71%
(12,8,4)	256	23.44%	23.44%	35.94%	35.94%	9.21%	118.26%
(12,8,8)	1028	36.38%	24.81%	39.79%	39.79%	0.10%	156.88%
(16,10,4)	496	18.95%	21.77%	37.70%	37.70%	8.28%	138.93%
(16,10,8)	1920	30.16%	21.98%	40.89%	40.89%	5.00%	179.00%
Average over all tested cases		35.05%	24.27%	36.63%	36.66%	4.81%	130.4%

which is the default compiler of the OS. During compilation, O3 optimization, SSE4 and AVX2 instruction sets are enabled. Using different compilers and different compiler options may yield slightly different coding throughputs, but will not change the relative relationship among different coding methods, when the same compiler and compiler options are used across them.

It can be seen that among the first four techniques, BN usually provides roughly 35% improvement over the baseline on average. The variation among different (n,k,w) parameter settings is not negligible, which is likely caused by the specific field chosen and the number of possible choices of (X,Y) coefficients in the Cauchy Reed-Solomon codes. In comparison, smart scheduling can provide a more modest $\sim 24\%$ improvement. Both versions of matching algorithms can also provide significant improvements over the baseline, however, there is not a clear winner between the two versions of the matching algorithms.

Between the latter two techniques, S-CO can provide a gain of $\sim$5%, while V-XOR is able to improve the coding speed by roughly $100\% - 200\%$. The improvement observed in our work by S-CO is considerably less significant compared to the 23-36% improvement reported by Luo et al. [12], which we suspect is due to the improvement in cache size and cache prediction algorithm in modern CPUs. Indeed, when we test the same approach on different operating systems and CPUs, different (sometimes significantly different) amounts of improvement can be observed. Among all the techniques, V-XOR appears to be able to provide the most significant performance improvement.

The performance of directly vectorizing finite field operations [16] is not included in the set of tests above, because it belongs to a completely different computation chain. It does not utilize the bitmatrix representation at all, and thus completely bypasses all other techniques. This observation in fact raises the following question: can vectorizing XORs within the bitmatrix framework be a better choice than vectoring finite field operations directly? Surprisingly, the question has not been answered in the existing literature. As we shall discuss in the next section, our result suggests that the answer to this question indeed appears to be positive.

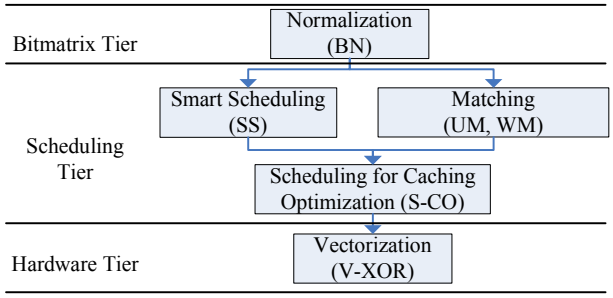

Figure 2: Operation tiers of the individual techniques

3.2 Combining the Individual Techniques

Equipped with individual improvements reported above, we are interested in whether and how these techniques can be combined to achieve more efficient erasure encoding. The techniques we test can be categorized into three tiers: the bitmatrix tier, the scheduling tier, and the hardware-related tier, as shown in Figure 2.

Since these techniques mostly reside in different tiers, they can be applied in tandem. The only exception is among the SS, UM, and WM techniques, since they are essentially optimizing the same component in the computation chain. As such, we need choose the technique or techniques to adopt. Additionally, although BN is able to provide improvement and can be applied together with other techniques, it is essentially also a procedure to optimize the bitmatrix, and the set of tests does not indicate whether it is still going to be effective when combined with SS, UM, or WM. The S-CO

Table 2: Combination of individual strategies

i	j
BN disabled (0)	no XOR reuse (0)
	SS (1)
BN enabled (1)	UM (2)
	WM (3)

technique is basically independent of the other techniques, and thus we can always invoke it for any combined strategies. Similarly, V-XOR can be applied directly together with all the other techniques. In fact, because of the significance of the improvement offered by V-XOR, including it should be a priority when using the techniques jointly.

We use a pair of indices (i, j) to enumerate the eight possible combinations of bitmatrix-based techniques, where $i \in \{0,1\}$ and $j \in \{0,1,2,3\}$, as shown in Table 3.2. For example, strategy-$(1,3)$ means both BN and WM are used. These combinations are the candidate strategies, within the bitmatrix framework, that we need to select from.

4 Selecting Coding Strategy under Optimized Bitmatrices

Our next task is to select one of eight strategies which can offer the best performance. The complication here is that since different choices of the (X,Y) vector in Cauchy Reed-Solomon code can lead to different computation load during erasure coding computation (see [19]), the (X,Y) coefficients need to be optimized. In other words, the strategies should be evaluated with such optimized bitmatrices. For this purpose, we first conduct heuristic optimizations to minimize the cost function of the total number of XOR and copy operations, under these eight strategies, the result of which is used to determine the best strategy. At the end of the section, we discuss possible improvement to the cost function.

4.1 Bitmatrix Optimization Algorithms

For a fixed choice of (X,Y) which determines the Cauchy parity coding matrix P, a given (i, j)-strategy will induce a given number of total computation operations, including XORs and copyings. Let us denote the function mapping from (X,Y) to this cost function as $c_{i,j}(X,Y)$. To compute, for example, $c_{1,2}(X,Y)$, we first conduct bitmatrix normalziation on the Cauchy matrix induced by (X,Y), then apply the unweighted matching algorithm to obtain the number of XOR operations and the number of copying operations in the encoding computation, and finally compute the total number of the operations. Our goal here is thus to find the choice of (X,Y) vector that minimizes this cost function. Due to the complex relation between the choice of (X,Y) vector and the cost function value, it is not possible to find the optimal solution using standard optimization techniques. Instead, we adopt two heuristic optimization procedures: simulated annealing (SA) and genetic algorithm (GA).

In the simulated annealing, there are several parameters that need to be set, however, we found that the results in this application is not sensitive to them. The only parameter of material importance is the annealing factor Δ, which control the rate of cooling.

For the genetic algorithm, we defined the population as a set of (X,Y) vectors. There is also a set of standard parameters in genetic algorithm (such as the crossover rate and the mutation rate), but the most important factor appears to be the crossover procedure in this setting. We considered two crossover methods to generate a child Cauchy matrix.

1. Random crossover: From the set of finite field elements which appear in the parent vectors (X,Y), choose $k+m$ elements at random and produce a random assignment of the new (X,Y) as the child.

2. Common elements first crossover: The finite field elements which appear in both parents are selected first as the element of child, and the others are chosen at random from the other elements which appear only in one of the parents.

The second approach tends to provide better new bitmatrices, which appears to match our intuition that some assignments are better than others, and keeping the good trait in the children may produce even better assignments.

In Table 3, we include a subset of (n,k,w) parameter choices we have attempted using the two optimization approaches together with the case when (X,Y) vector is assigned according to the sequential order in the finite field; the other test results are omitted due to space constraint. It can be seen that the genetic algorithm provides better solutions than simulated annealing in most cases, and for this reason we shall adopt GA in the subsequent discussion.

Due to the heuristic nature of the two algorithms, the readers may question whether the optimized (X,Y) choice can indeed provide any performance gain. It can be seen in Table 3, that both SA and GA can provide significant improvement on the total number of operations by finding good bitmatrices. In Figure 3, we further plot the amounts of cost reduction for different (n,k,w) parameters, from the baseline approach of without any optimization. It can be seen for most (n,k,w) parameters, meaningful (sometime significant) gains of $\sim 5\%$ to $\sim 25\%$ can be obtained. Thus, although the two heuristic optimization methods cannot guarantee finding the optimal solutions, they do lead to considerably improved bitmatrix choices.

4.2 Choosing the Best (i, j)-Strategy

We can now select the best (i, j)-strategy, using the optimized bitmatrices obtained by the genetic algorithm. In Figure 4, the cost function values of different (i, j)-strategies

Table 3: Comparison of the total number of XOR and copy operations for all (i,j)-strategies, when the bitmatrices are obtained without optimization, by simulated annealing, and by the genetic algorithm, respectively, as the three columns in each box.

(i,j)	$(n,k,w)=(8,6,4)$			$(n,k,w)=(9,6,4)$			$(n,k,w)=(10,6,4)$			$(n,k,w)=(12,8,4)$			$(n,k,w)=(16,10,4)$		
(0,0)	112	**77**	**77**	164	122	**117**	216	173	**162**	272	**232**	**232**	520	**462**	464
(0,1)	94	70	**68**	134	102	**100**	172	137	**134**	212	190	**188**	412	**368**	**368**
(0,2)	90	72	**68**	127	103	**101**	164	134	**132**	204	186	**180**	376	**344**	345
(0,3)	90	72	**68**	127	102	**101**	164	**132**	**132**	204	184	**182**	376	**343**	345
(1,0)	68	**58**	**58**	114	99	**98**	161	142	**141**	212	**198**	**198**	426	**419**	**419**
(1,1)	64	**58**	**58**	99	90	**89**	138	**120**	**120**	189	174	**171**	365	**352**	**352**
(1,2)	64	58	**57**	98	**87**	**87**	133	**118**	**118**	176	165	**164**	326	317	**316**
(1,3)	64	58	**57**	98	90	**87**	132	**118**	**118**	175	**164**	**164**	326	**316**	**316**
(i,j)	$(n,k,w)=(8,6,8)$			$(n,k,w)=(9,6,8)$			$(n,k,w)=(10,6,8)$			$(n,k,w)=(12,8,8)$			$(n,k,w)=(16,10,8)$		
(0,0)	378	291	**247**	573	467	**425**	768	611	**582**	1060	880	**841**	1968	1685	**1681**
(0,1)	256	234	**225**	413	393	**349**	556	518	**479**	805	740	**684**	1546	1432	**1374**
(0,2)	286	227	**217**	408	346	**321**	532	474	**450**	726	636	**591**	1304	1180	**1170**
(0,3)	286	216	**197**	408	357	**329**	532	460	**450**	726	636	**627**	1304	1180	**1170**
(1,0)	185	151	**142**	328	286	**262**	467	434	**419**	686	613	**563**	1389	1293	**1246**
(1,1)	164	155	**133**	285	270	**230**	411	383	**371**	593	557	**544**	1264	1184	**1133**
(1,2)	167	143	**134**	272	244	**225**	377	343	**335**	520	487	**462**	998	951	**922**
(1,3)	167	144	**130**	273	249	**233**	377	**337**	**337**	520	473	**467**	995	941	**932**

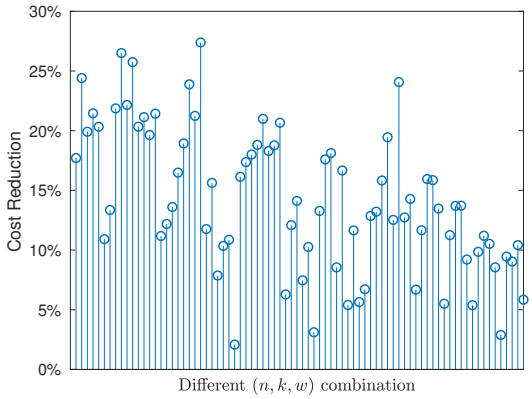

Figure 3: Cost reductions obtained by the genetic algorithm for different (n,k,w) parameters (sorted by n).

are shown, under various (n,k,w) parameters. Here again we have chosen a subset of representative test results, and omit others due to space constraint. It can be seen that the strategies $(1,2)$ and $(1,3)$ are the best among all the possibilities, and they do not show any significant difference between themselves.

4.3 Refining the Cost Function

We have so far used the total number of XORs and copying operations as the cost function in the optimization of bitmatrices. However, this choice may not accurately capture all the computation operations, and as such, we next consider three possible cost functions: 1) the total number of XORs, 2) the total number of operations, including XORs and copyings, and 3) a weighted combination of the number of XORs and that of copying operations. In the last option, we set the weight according to the empirical testing result of these operation on the target workstation: the time taken for copying (memcpy) and that for XORing the same amount of data are measured. On our platform, the weight given to XOR is roughly 1.5 the weight given to memory copying. To distinguish from the cost function $c_{i,j}(X,Y)$, we write this last cost function as $c'_{i,j}(X,Y)$.

The effectiveness of these three cost functions is evaluated and shown in Table 4 by using the genetic algorithm to find the optimized bitmatrices. The resulting bitmatrices obtained under the three cost functions are used to encode the data with the $(1,3)$-strategy, and we compare the encoding throughput values. It can be seen that the third cost function is able to most accurately capture the encoding computation cost in practice. The improvements obtained by the refined cost function $c'_{i,j}(X,Y)$, in most cases, are not extremely large, ranging from $0\%-10\%$, and occasionally it does cause a minor performance degradation than the cost function $c_{i,j}(X,Y)$.

5 The Proposed Design and Coding Procedure, and Performance Evaluation

From the previous discussion, the proposed bitmatrix design procedure is quite clear: perform a suitable optimization algorithm (the genetic algorithm is used in this work) with the weighted cost function $c'_{1,2}(X,Y)$ or $c'_{1,3}(X,Y)$. The proposed erasure coding procedure then naturally involves the

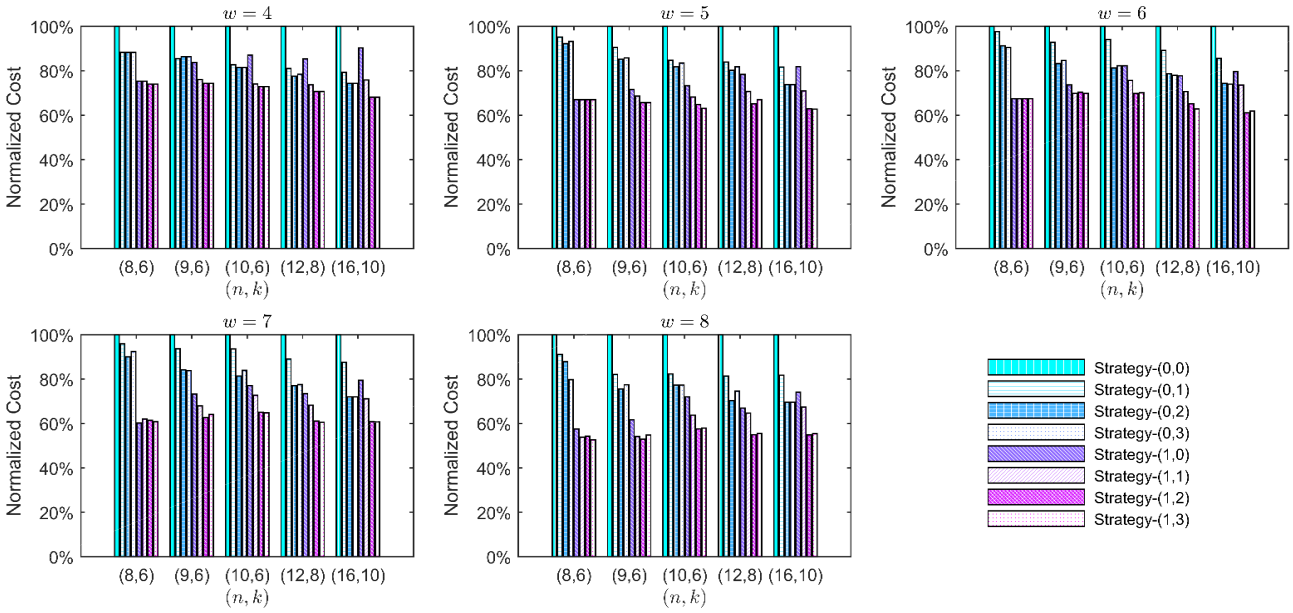

Figure 4: Total number of XOR and copying operations for all (i, j) strategies with optimized bitmatrices

Table 4: Encoding throughput (GB/s) using bitmatrices obtained by the genetic algorithm under different cost functions

(n,k,w)	Cost Function		
	# of XOR	# of XOR and copying	Weighted
(8,6,4)	4.64	4.66	**4.68**
(8,6,8)	4.30	**4.35**	4.32
(9,6,4)	3.72	3.73	**3.80**
(9,6,8)	3.28	3.28	**3.44**
(10,6,4)	2.33	2.51	**2.52**
(10,6,8)	1.99	1.97	**2.11**
(12,8,4)	2.96	3.11	**3.16**
(12,8,8)	2.54	2.56	**2.58**
(16,10,4)	2.29	2.29	**2.32**
(16,10,8)	1.71	1.72	**1.74**

corresponding components: from the selected (X,Y), produce the corresponding bitmatrix by bitmatrix normalization, then generate the computation schedule from the produced bitmatrix using the selected matching algorithm and following the cache-friendly order, and perform the vectorized XOR operations using the necessary CPU instructions.

In the sequel, we discuss a few details in integrating these techniques, and then provide performance evaluation in comparison to the existing approaches.

5.1 Integrating XOR Matching and S-CO

As described in 2.3.2, ordering the encoding operation sequence according to the data order can increase spatial data locality, which helps reduce cache miss penalty. The schedules in [12] contain only data to parity operations

$$u_i \rightarrow p_j,$$

where the arrow indicates the direction of data flow for either a memory copy or an XOR operation. Because of the UM procedure or the WM procedure in the computation chain, the common XOR pairs need to be computed and stored as intermediate results, denoted as int_l. As such, in the proposed procedure, the schedule contains three types of operations

$$u_i \rightarrow p_j, \quad u_i \rightarrow int_l, \quad int_l \rightarrow p_j.$$

To reduce cache misses, we need to extend the ordering method to handle these three cases. This can be accomplished by following the sequential order of (XORing and copying) the data bits to the parity bits or intermediate bits first, then the intermediate bits to the parity bits, e.g.,

$$\begin{bmatrix} u_0 \rightarrow p_0 \\ u_0 \rightarrow int_0 \\ \cdots \\ u_1 \rightarrow p_3 \\ \cdots \\ int_0 \rightarrow p_4 \\ \cdots \end{bmatrix}.$$

This order ensures that each data bit u_i will be read exactly once, which maintains the spatial data locality.

5.2 Vectorizing XOR Operations

Each step in the computation schedule is either a copying operation, or an XOR operation. In practice, instead of performing them in a bit-wise manner, a set of bits is processed at a time, i.e., using an extended word format. The smallest such extension is a byte (8bits) in computer systems, and similarly a long long word has 64 bits. For CPUs with an SIMD instruction set, the extended word can be 128 bits, 256 bits, or 512 bits. A single instruction thus completes the operation on 8 bits, 64 bits, 128 bits, 256 bits, or 512 bits, respectively. The instructions we used in this work are included in Intel® Intrinsics instruction set, which has also been adopted in AMD CPUs.

The C code to perform the XOR operation is as follows:

```
#include <x86intrin.h>
void fast_xor( char *r1,   /* region 1 */
               char *r2,   /* region 2 */
               char *r3,   /* r3 = r1 ^ r2 */
               int size)   /* bytes of region */
{
    __m128i *b1, *b2, *b3;
    int vec_width = 16,j;
    int loops = size / vec_width;
    for(int j = 0;j<loops;j++)
    {
        b1 = (__m128i *)(r1 + j*vec_width);
        b2 = (__m128i *)(r2 + j*vec_width);
        b3 = (__m128i *)(r3 + j*vec_width);
        *b3 = _mm_xor_si128(*b1, *b2);
    }
}
```

The SSE data type `__m128i` is a vector of 128 bits, which can be easily converted from any common data types such as `char`, `int`, or `long`. The instruction `_mm_xor_si128` computes the bitwise XOR of 128 bits. To utilize the 256 bits AVX2 instructions or the 512 bits AVX-512 instructions, the migration is rather straightforward in the proposed computation procedure as follows:

1. Update the data format:
   AVX2 : `__m128i` → `__m256i`
   AVX-512: `__m128i` → `__m512i`

2. Update the bitwidth parameter:
   AVX2 : `vec_width = 16;` → `vec_width = 32;`
   AVX-512: `vec_width = 16;` → `vec_width = 64;`

3. Update the instruction:
   AVX2 : `_mm_xor_si128` → `_mm256_xor_si256`
   AVX-512: `_mm_xor_si128` → `_mm512_xor_epi32`

In our experience, performing the XOR operation on the same amount of data with 64 bitwidth (long long format) is 30% slower than `_mm_xor_si128`, and it is 50% slower than `_mm256_xor_si256`. Note that our tests are performed on an AMD CPU, however INTEL CPUs may have somewhat different characteristics. The instruction `_mm512_xor_epi32` is expected to be even faster, however we currently do not have such a platform for testing.

Table 5: Encoding throughput (GB/s) for methods that allow general (n,k) parameters and $w = 8$

(n,k)	Proposed	Vectorized XOR-based CRS code	Vectorized GF-based RS code [16]
(7,5)	4.64	**4.73**	2.52
(8,6)	5.21	**5.22**	2.70
(9,7)	5.32	**5.45**	2.74
(10,8)	5.36	**5.59**	2.77
(12,10)	5.72	**5.88**	2.81
(8,5)	**3.19**	2.75	1.76
(9,6)	**3.49**	2.84	1.77
(10,7)	**3.67**	2.79	1.80
(11,8)	**3.72**	2.92	1.82
(13,10)	**3.82**	3.10	1.84
(10,6)	**2.55**	2.15	1.31
(11,7)	**2.75**	2.17	1.32
(12,8)	**2.86**	2.20	1.35
(14,10)	**2.86**	2.19	1.40
(15,10)	**2.30**	1.79	1.11
(16,10)	**1.96**	1.48	0.92

5.3 Encoding Performance Evaluation

Here we provide comprehensive encoding throughput test results between the proposed approach and several well-known efficient erasure coding methods in the literature, as well as the erasure array codes designed for high throughput. The latter class includes EVENODD code [2], RDP code [6], Linux Raid-6 [14], STAR code [8], and Quantcast-QFS [13] code. EVENODD code, RDP code, Raid-6 are specially designed to have two parities, and STAR code and Quantcast-QFS are specially designed to have only three parities; in order to make the comparison fair, we use 128-bit vectorized XOR discussed in Section 5.2 for these codes as well. Since open source implementations for these codes are not available, we have implemented these coding procedures, with and without vectorization, to use in our comparison. The former class of codes includes several efficient Cauchy Reed-Solomon code implementations based on bitmatrices (XOR-based CRS) [12, 16, 18], and finite field vectorized Reed-Solomon code (GF-based RS code) [16]; the source code for them can be found in the Jerasure library 2.0 [16] publicly available online, which is used in our comparison. The implementation in Jerasure library 2.0 is based on vectorizing (through 128-bit instruction) finite field operation in $GF(2^8)$, $GF(2^{16})$, and $GF(2^{32})$. The Cauchy Reed-Solomon code implementation in Jerasure library 1.2 [18] can be adapted to utilize with XOR-level vectorization, however, it would

Table 6: Encoding throughputs (GB/s): Three parities

(n,k,w)	Proposed	Vectorized XOR-based CRS code	STAR code [8]	Quancast QFS [13]
(8,5,4)	**3.59**	3.25	2.97	2.92
(8,5,8)	**3.19**	2.75	2.97	2.92
(9,6,4)	3.52	**3.72**	3.42	3.04
(9,6,8)	**3.49**	2.84	3.42	3.04
(10,7,4)	**4.15**	3.86	3.76	3.25
(10,7,8)	3.67	2.79	**3.76**	3.25
(11,8,4)	**4.36**	4.13	3.94	3.27
(11,8,8)	3.72	2.92	**3.94**	3.27
(13,10,4)	**4.51**	4.08	4.37	3.41
(13,10,8)	3.82	3.10	**4.37**	3.41

Table 7: Encoding throughputs (GB/s): Two parities

(n,k,w)	Proposed	Vectorized XOR-based CRS code	Vectorized Raid-6	EVEN ODD [2]	RDP [6]
(7,5,4)	**5.16**	4.85	n/a	4.28	4.37
(7,5,8)	4.64	**4.73**	2.18	4.28	4.37
(8,6,4)	4.67	**5.22**	n/a	4.83	4.95
(8,6,8)	5.21	**5.22**	2.15	4.83	4.95
(9,7,4)	**5.77**	5.59	n/a	5.20	5.32
(9,7,8)	5.32	**5.45**	2.16	5.20	5.32
(10,8,4)	**5.90**	5.23	n/a	5.50	5.69
(10,8,8)	5.36	5.59	2.17	5.50	**5.69**
(12,10,4)	6.23	6.00	n/a	6.02	**6.24**
(12,10,8)	5.72	5.88	2.17	6.02	**6.24**

not include the UM (or WM) component and the refinement of the cost function discussed in Section 4.3.

In the tests reported below, the parameters k varies from 5 to 10, m from 2 to 6, and w from 4 to 8. Some of these parameters do not apply for some of the reference codes and coding methods, which will be indicated as n/a in the result tables. The comparison is first presented in three groups.

- In Table 5, the proposed approach is compared with vectorized XOR-based Cauchy Reed-Solomon code, and vectorized finite field Reed-Solomon code, when $w = 8$. All three approaches are applicable for general (n,k) coding parameters, however the implementation of vectorized finite field operations in [16] can only use $w = 8, w = 16$ or $w = 32$; in contrast, the other two approaches can use other w values. Here we choose $w = 8$ for a fair comparison. When $m = n - k = 2$, it is seen that vectorized XOR-based Cauchy Reed-Solomon code is slightly faster than the proposed approach, because the SS technique in these cases in fact provides a slighter better scheduling than WM. When m is larger than 2, the proposed procedure can provide a more significant throughput advantage. Vectorizing finite field operations is always the worse choice among the three by a large margin.

- In Table 6, the proposed approach is compared with well-known codes with three parities. It is seen that the proposed approach is able to compete with these coding theory based techniques. It should be noted that STAR code and Quantcast QFS code do not rely on the parameter w, and thus the throughput performances for $w = 4$ and $w = 8$ are the same for each (n,k) parameter.

- In Table 7, the proposed approach is compared with well-known codes with two parities. It is again seen that the proposed approach is able to compete with these established coding techniques. EVENODD code and RDP code do not rely on the parameter w, and thus the throughput performances for $w = 4$ and $w = 8$ are the same.

Table 8: Encoding throughput improvements over references

Reference codes or methods	Improvement by proposed code
General (n,k) Codes	
GF-based RS code w/o vectorization	552.27%
XOR-based CRS code w/o vectorization	53.65%
Vectorized GF-based RS code [16]	99.82%
Vectorized XOR-based CRS code	14.98%
Three Parities Codes	
STAR [8]	5.59%
Quancast-QFS [13]	21.68%
Two Parities Codes	
Raid-6 w/o vectorization	206.88%
Vectorized Raid-6	142.07%
RDP [6]	5.85%
EVENODD [2]	8.79%

In Table 8, we list the amounts of improvements of the proposed approach over other reference approaches or codes, averaged over all tested (n,k,w) parameters. It is seen that the proposed approach can provide improvements over all existing techniques, some by a large margin. The result in this table is included here to provide a summary on the performance by various techniques, however for individual (n,k,w) parameter, the performance may vary as indicated by the previous three tables.

5.4 Decoding Performance Evaluation

In practical systems, data is usually read out directly without using the parity symbols, unless the device storing the data symbols becomes unavailable, i.e., in the situation of degraded read. Therefore, the most time consuming computation in erasure code decoding is in fact invoked much less often, which implies that the decoding performance should be viewed as of secondary importance. However, it is still

Table 9: Decoding throughput (GB/s) for methods that allow general (n,k) parameters and $w = 8$

(n,k)	Proposed	Vectorized XOR-based CRS code	Vectorized GF-based RS code [16]
(7,5)	3.87	**4.56**	2.58
(8,6)	**5.45**	4.86	2.67
(9,7)	4.46	**5.06**	2.70
(10,8)	4.89	**5.11**	2.75
(12,10)	4.45	**5.52**	2.79
(8,5)	**3.04**	2.11	1.71
(9,6)	**2.94**	2.20	1.74
(10,7)	**3.28**	2.29	1.76
(11,8)	**3.08**	2.31	1.71
(13,10)	**3.21**	2.37	1.88
(10,6)	**2.38**	1.80	1.31
(11,7)	**2.35**	1.85	1.32
(12,8)	**2.54**	1.87	1.33
(14,10)	**2.47**	1.89	1.38
(15,10)	**2.00**	1.48	1.09
(16,10)	**1.77**	1.30	0.91

Table 10: Decoding throughputs (GB/s): Three parities

(n,k,w)	Proposed	Vectorized XOR-based CRS code	STAR code [8]	Quancast QFS [13]
(8,5,4)	**4.28**	3.08	3.20	1.77
(8,5,8)	**3.04**	2.11	3.20	1.77
(9,6,4)	**4.13**	3.41	3.23	1.74
(9,6,8)	**2.94**	2.20	3.23	1.74
(10,7,4)	**4.55**	3.53	3.52	1.77
(10,7,8)	**3.28**	2.29	3.52	1.77
(11,8,4)	**4.70**	3.78	3.13	1.68
(11,8,8)	**3.08**	2.31	3.13	1.68
(13,10,4)	**4.86**	3.71	3.50	1.71
(13,10,8)	3.21	2.37	**3.50**	1.71

important to understand the impact of optimizing the encoding bitmatrix and procedure, which was our main focus. In this section, we present the decoding performance of various methods, along the similar manner as for the encoding performance. Only the performance for the worst case failure pattern (the most computationally expensive case) is reported, when m data symbols are lost.

As seen in Table 9, the proposed approach can provide better decoding throughput comparing to vectorized XOR-based Cauchy Reed-Solomon code and vectorized GF-based RS code, except for some cases when $m = 2$. For codes with three parities, it can be seen from Table 10 that the decoding throughput of the proposed approach still outperforms well-known codes in the literature specifically designed for this case. For codes with two parities, as shown in Table 7, the decoding throughput of proposed approach is usually lower than EVEN-ODD and RDP codes.

Table 11: Decoding throughputs (GB/s): Two parities

(n,k,w)	Proposed	Vectorized XOR-based CRS code	Vectorized Raid-6	EVEN ODD [2]	RDP [6]
(7,5,4)	5.52	4.85	n/a	6.66	**7.28**
(7,5,8)	3.87	4.65	2.64	6.66	**7.28**
(8,6,4)	5.43	5.14	n/a	7.42	**8.00**
(8,6,8)	5.45	4.86	2.67	7.42	**8.00**
(9,7,4)	6.03	5.37	n/a	7.65	**8.13**
(9,7,8)	4.46	5.06	2.73	7.65	**8.13**
(10,8,4)	5.88	5.73	n/a	7.93	**8.44**
(10,8,8)	4.89	5.11	2.77	7.93	**8.44**
(12,10,4)	6.23	5.89	n/a	7.49	**9.10**
(12,10,8)	4.45	5.52	2.81	7.49	**9.10**

In summary, the optimized encoding procedure we propose does not appear to significantly impact the performance of the decoding performance in most cases (when $m \geq 3$), which itself is a less important performance measure in practice than the encoding performance that we focus on in this work.

6 Conclusion

We performed a comprehensive study of the erasure coding acceleration techniques in the literature. A set of tests was conducted to understand the improvements and the relation among these techniques. Based on these tests, we consider combining the existing techniques and jointly optimize the bitmatrix. The study led us to a simple procedure: produce a computation schedule based on an optimized bitmatrix (using a cost function matching the computation strategy and workstation characteristic), together with the BN and WM (or UM) technique, then use vectorized XOR operation in the computation schedule. The proposed approach is able to provide improvement over most existing approaches, particularly when the number of parity is greater than two. One particularly important insight of our work is that vectorization at the XOR-level using the bitmatrix framework is a much better approach than vectorization of the finite field operations in erasure coding, not only because of the better throughput performance, but also because of the simplicity in migration to new generation CPUs.

References

[1] Library for efficient modeling and optimization in networks. `http://lemon.cs.elte.hu/trac/lemon`, 2003.

[2] Mario Blaum, Jim Brady, Jehoshua Bruck, and Jai Menon. EVENODD: An efficient scheme for tolerat-

ing double disk failures in RAID architectures. *IEEE Transactions on computers*, 44(2):192–202, 1995.

[3] Johannes Blomer, Malik Kalfane, Richard Karp, Marek Karpinski, Michael Luby, and David Zuckerman. An XOR-based erasure-resilient coding scheme. Technical Report TR-95-048, University of California at Berkeley, 1995.

[4] Kevin D. Bowers, Ari Juels, and Alina Oprea. Hail: A high-availability and integrity layer for cloud storage. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 187–198. ACM, 2009.

[5] Peter M. Chen, Edward K. Lee, Garth A. Gibson, Randy H. Katz, and David A. Patterson. RAID: High-performance, reliable secondary storage. *ACM Computing Surveys (CSUR)*, 26(2):145–185, 1994.

[6] Peter Corbett, Bob English, Atul Goel, Tomislav Grcanac, Steven Kleiman, James Leong, and Sunitha Sankar. Row-diagonal parity for double disk failure correction. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, pages 1–14, 2004.

[7] Cheng Huang, Jin Li, and Minghua Chen. On optimizing XOR-based codes for fault-tolerant storage applications. In *Proceedings of Information Theory Workshop*, pages 218–223, 2007.

[8] Cheng Huang and Lihao Xu. STAR: An efficient coding scheme for correcting triple storage node failures. *IEEE Transactions on Computers*, 57(7):889–901, 2008.

[9] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, et al. Oceanstore: An architecture for global-scale persistent storage. In *ACM SIGARCH Computer Architecture News*, volume 28, pages 190–201. ACM, 2000.

[10] Shu Lin and Daniel J. Costello. *Error control coding*. Pearson Education India, 2001.

[11] Witold Litwin and Thomas Schwarz. LH*RS: A high-availability scalable distributed data structure using Reed Solomon codes. In *Proceedings of the ACM SIGMOD Record*, volume 29, pages 237–248, 2000.

[12] Jianqiang Luo, Mochan Shrestha, Lihao Xu, and James S. Plank. Efficient encoding schedules for XOR-based erasure codes. *IEEE Transactions on Computers*, 63(9):2259–2272, 2014.

[13] Michael Ovsiannikov, Silvius Rus, Damian Reeves, Paul Sutter, Sriram Rao, and Jim Kelly. The Quantcast file system. *Proceedings of the VLDB Endowment*, 6(11):1092–1101, 2013.

[14] James S. Plank. The RAID-6 liberation code. In *Proceedings of the 6th Usenix Conference on File and Storage Technologies*, pages 97–110, 2008.

[15] James S. Plank. XOR's, lower bounds and MDS codes for storage. In *2011 IEEE Information Theory Workshop (ITW)*, pages 503–507, 2011.

[16] James S. Plank, Kevin M. Greenan, and Ethan L. Miller. Screaming fast Galois field arithmetic using intel SIMD instructions. In *Proceedings of the 11st Usenix Conference on File and Storage Technologies*, pages 299–306, 2013.

[17] James S. Plank, Catherine D. Schuman, and B. Devin Robison. Heuristics for optimizing matrix-based erasure codes for fault-tolerant storage systems. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*, pages 1–12, 2012.

[18] James S. Plank, Scott Simmerman, and Catherine D. Schuman. Jerasure: A library in C/C++ facilitating erasure coding for storage applications-version 1.2. Technical Report CS-08-627, University of Tennessee, 2008.

[19] James S. Plank and Lihao Xu. Optimizing Cauchy Reed-Solomon codes for fault-tolerant network storage applications. In *Proceedings of Fifth IEEE International Symposium on Network Computing and Applications*, pages 173–180, 2006.

[20] Irving S. Reed and Gustave Solomon. Polynomial codes over certain finite fields. *Journal of the society for industrial and applied mathematics*, 8(2):300–304, 1960.

[21] Hakim Weatherspoon and John D. Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. In *Proceedings of the International Workshop on Peer-to-Peer Systems*, pages 328–337, 2002.

[22] Zooko Wilcox-O'Hearn and Brian Warner. Tahoe–The least-authority filesystem. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 21–26, 2008.