ERASURE CODING OPTIMIZATION FOR DATA STORAGE: ACCELERATION TECHNIQUES AND DELAYED PARITIES GENERATION

A Dissertation

by

TIANLI ZHOU

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

May 2020

Major Subject: Computer Engineering

ABSTRACT

Various techniques have been proposed in the literature to improve erasure code computation efficiency, including optimizing bitmatrix design and computation schedule, common XOR operation reduction, caching management techniques, and vectorization techniques. These techniques were largely proposed individually, and in this work, we seek to use them jointly. To accomplish this task, these techniques need to be thoroughly evaluated individually, and their relation better understood. Building on extensive testing, 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 computation cost function, and also a straightforward coding procedure: follow a computation schedule produced from the optimized bitmatrix to apply XOR-level vectorization. This procedure provides better performances than most existing techniques (e.g., those used in ISA-L and Jerasure libraries), and sometimes can even compete against well-known but less general codes such as EVENODD, RDP, and STAR codes. One particularly important observation is that vectorizing the XOR operations is a better choice than directly vectorizing finite field operations, not only because of the flexibility in choosing finite field size and the better encoding throughput, but also its minimal migration efforts onto newer CPUs.

A delayed parity generation technique for maximum distance separable (MDS) storage codes is proposed as well, for two possible applications: the first is to improve the write-speed during data intake where only a subset of the parities are initially produced and stored into the system, and the rest can be produced from the stored data during a later time of lower system load; the second is to provide better adaptivity, where a lower number of parities can be chosen initially in a storage system, and more parities can be produced when the existing ones are not sufficient to guarantee the needed reliability or performance. In both applications, it is important to reduce the data access as much as possible during the delayed parity generation procedure. For this purpose, we first identify the fundamental limit for delayed parity generation through a connection to the well-known multicast network coding problem, then provide an explicit and low-complexity code transformation that is applicable on any MDS codes to obtain optimal codes. The problem we consider is closely related to the regenerating code problem, however the proposed codes are much simpler and have a much smaller subpacketization factor than regenerating codes, and thus our result in fact shows that blindly adopting regenerating codes in these two settings is unnecessary and wasteful.

Moreover, two aspects of this approach is addressed. The first is to optimize the underlying coding matrix, and the second is to understand its behavior in a system setting. For the former, we generalize the existing approach by allowing more flexibility in the code design, and then optimize the underlying coding matrix in the familiar bitmatrix-based coding framework. For the latter, we construct a prototype system, and conduct tests on a local storage network and on two virtual machine based setups. In both cases, the results confirm the benefit of delayed parity generation when the system bottleneck is in the communication bandwidth instead of the computation.

DEDICATION

To my beloved wife Shan Jiang, my son Youyan, and my parents.

To my missing grandfathers and grandmothers.

ACKNOWLEDGMENTS

I would like to first express my thanks to my incomparable and patient supervisor, Dr. Chao Tian, who has supported me throughout my entire study of Ph.D. program. I am extremely grateful for his guidance of my research as well as personal support of my life.

I would like to appreciate other professors and colleagues in Texas A&M University and University of Tennessee, Knoxville, which includes but not limited to Dr. James Plank, Dr. Michael A. Langston, Kai Zhang, Lei Zheng, Dr. Ting Zang, Qiang Zhang, Dr. Tao Guo, Ruida Zhou, Wenjing Chen, Hao Ye, Ramin Nabati, Sara Mousavi, Katie Bryan and Vickie Winston.

Last but not least, my education and life in China established my background and the foundation of my doctoral study, which I would also convey my deepest gratitude to all the advisers and mates who provided significant help to my study and life: Mr. Yao Feng, Dr. Guolong Liang, Dr. Yilin Wang, Dr. Jidan Mei, Dr Jin Fu, Mr. Longbin Su, Dr. Xianjun Pei and Dr. Nan An.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supported by a dissertation committee consisting of Professor Chao Tian [advisor], Professor Stavros Kalafatis and Professor Alexander Sprintson of the Department of Electrical and Computer Engineering and Professor Anxiao (Andrew) Jiang of the Department of Computer Science and Engineering.

The derivation of fundamental limit for Chapter 4 was provided by Professor Chao Tian, which was published in 2018 in IEEE International Symposium on Information Theory. The code construction of delay parity generation in Chapter 4 is collaborated with Sara Mousavi from the University of Tennessee, Knoxville.

All other work conducted for the dissertation was completed by the student independently.

Funding Sources

Graduate study was supported by a graduate assistantship from Texas A&M University.

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE Page

LIST OF TABLES

TABLE Page

1. INTRODUCTION AND LITERATURE REVIEW [1] [2]

1.1 Erasure Coding

The exploding amount of digital data being produced by users and collected by various source in this information age poses significant challenges to modern large-scale data storage systems. As the scale of data storage system grows, storage device failures or other system breakdowns become more and more common. System designers must plan ahead and build sufficient fault-tolerance capability in the system, in order to effectively deal with such uncertainties.

This fault-tolerance requirement in data storage systems implies that redundancy must be introduced in the stored data.

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 [6], to distributed storage systems [7, 8], and cloud storage systems [9]. The root of erasure codes can be traced back to the well-known Reed-Solomon codes [10], or more generally, maximum distance separable codes [11]. Roughly speaking, 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 such example is Quantcast File System (QFS) [12], 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.

Erasure codes have been recognized as a viable solution to replace simple data replication in distributed data storage systems (*e.g.*, [13, 14]), because they can reduce storage overhead, and at the same time provide superior data reliability. As data intensive applications, *i.e.*, big data analytics, become mainstream in both industrial and academic settings (*e.g.,* [15, 16, 17]), large

[1] ©2018 IEEE. Partly Reprinted, with permission, from S. Mousavi, T. Zhou and C. Tian, "Delayed parity generation in MDS storage codes", 2018 IEEE International Symposium on Information Theory (ISIT), June 2018

[2] ©2020 ACM. Partly Reprinted, with permission, from T. Zhou and C. Tian. 2020. "Fast Erasure Coding for Data Storage: A Comprehensive Study of the Acceleration Techniques". ACM Trans. Storage 16, 1, Article 7 (March 2020), 24 pages. DOI:https://doi.org/10.1145/3375554

scale data storage systems (sometimes referred to as cloud storage) based on erasure codes are gaining more and more momentum [18, 12, 19], and existing solutions are also incorporating erasure codes in their new or future releases [20, 21].

The parity data written in the system is generated from the raw information data, which needs to be transported and written onto the individual storage devices. In contrast, for systems without any data redundancy (no protection), only the raw information data needs to be transported and written. This additional amount of data induces more traffic and more disk IO, and the write speed is consequently slower in such systems than systems without any data redundancy. One question that we are interested in is whether the data write speed can be improved in such erasure-coded data storage systems.

1.2 Acceleration Techniques

It has long been recognized that encoding data into its erasure-coded form will incur a much heavier computation load than simple data replication [22], 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 [23, 5, 24, 25, 26, 2], or to accelerate the computation by better utilizing the resources in modern CPUs [27, 28]. However, the relationship and possible joint optimization among these techniques is still an unanswered question.

As part of this dissertation, I'll find out the relationship among these accelerating techniques, as well as jointly optimized for better coding throughput, especially encoding throughput.

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 [29], Plank et al. [2] proposed efficient methods to encode using the "bitmatrix" representation. Several techniques were introduced in the same work to reduce the number of the XORs 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. [26] (see also the Liberation codes [30] 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 XORs along these lines were investigated by Plank et al. [31], and lower bounds (albeit rather loose in most cases) on the total number of XORs have also been found [32].

Though with the same goal of reducing the computation load in mind, the coding theory community addresses the issue from another perspective, where specific code constructions have been proposed. Several notable examples of such codes can be found in [23, 5, 24, 25]. 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 throughput 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), often 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 [28]. Also related to this approach of optimizing resource utilization, Luo et al. [27] 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, which often make the IO bandwidth the constraining factor, 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 for service providers. For example, the three-parity erasure code encoding function and decoding functions in the open-source Quantcast QFS system [12] were heavily optimized, and its high coding efficiency is viewed as one of its main advantages over competing systems. Secondly, 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. Thirdly, more advanced storage devices such as SSD have vastly improved IO performance, whose bandwidth may soon compete with the CPU computation throughput. Last but not least, reducing computation cost can also lead to direct reduction in energy consumption, which can contribute to significant savings in the operating cost in large scale systems. 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 questions, we discovered a particularly effective approach to accelerate erasure encoding: selecting bitmatrices optimized for the weighted sum of the number of XOR and copy operations, taking into consideration of the reduction from the common XOR chains, then using XOR-level vectorization for hardware acceleration. Within the proposed strategy, the packet size selected in the coding procedure can impact the coding performance significantly, and we also identify a simple strategy to determine its (almost) optimal value. A simple decoding procedure is also proposed with such optimized packet size in consideration. Extensive evaluation tests were then conducted on multiple CPU platforms, which show that the coding procedure we propose can provide significant improvement on the coding throughput compared to the existing approaches, such as those used in Jerasure libraries [2, 27, 28] and ISA-L library [33], ranging from 20% to 1000% under general coding parameters. Moreover, in some cases, the proposed approach can compete with the well-known EVENODD code [23], RAID-6 code [30], RDP code [24], STAR code [25], and triple-parity Reed-Solomon code in Quantcast-QFS [12], which were specifically designed for fast encoding and only for restricted parameters.

One particularly important observation we make in this work is that instead of vectorizing the finite field operation directly, which was made popular by Jerasure 2.0 [28] and also adopted by the open source library ISA-L [33], we should vectorize the XOR operation based on the bitmatrix representation. In the bitmatrix representation, different binary extension fields can be vectorized in a similar manner, whereas vectorizing finite field operations usually requires the field size to have certain special relation with the computer byte size, which is part of the reason that only $GF(2^8), GF(2^{16}), GF(2^{32})$ are allowed in Jerasure 2.0 and only $GF(2^8)$ is supported in ISA-L. Since operation in a smaller finite field is usually simpler than in a larger field, allowing flexible finite field size directly translates to throughput improvement. This advantage, together with the better optimized computation chain, leads to the overall throughput advantage. In addition, this approach has an important practical advantage: vectorizing general finite field operations involves specially designed algorithms for different field sizes based on the availability of the CPU-specific vectorization instructions, and thus the implementation requires a larger set of relevant operations using such instructions; in contrast, vectorizing XOR operations essentially involves only one instruction, which can be done either with the vectorized XOR instruction, or to some extent by delegating this vectorization task to the compiler, i.e., through the compiler auto-vectorization function. 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 completely hardware-agnostic.

1.3 Delayed Parity Generation

Another coding technique for erasure codes, referred to as *delayed parity generation (DPG)*, is defined where only a subset of the parity symbols are produced during the initial data intake, with the rest to be produced from the stored data at a later time. This technique is applicable in two scenarios. The first scenario is when the initial data write speed is critically important, and

DPG can be used to improve it by offloading part of the computation and transmission, hopefully to a time of lower system load. The second scenario is when there is uncertainty on the suitable number of parities to be used in the storage code in the system; the system administrator may wish to choose a small number initially to speed up system deployment and reduce cost, and only produce more parities when it turns out, after a period of system operation, these existing ones are insufficient to guarantee the required reliability or performance.

It is beneficial and important to reduce data access as much as possible during delayed parity generation. A naive approach is to download the raw information data completely and then generate the additional parities, which is clearly not efficient. In this work, we investigate MDS codes with optimal data access for DPG. It turns out that the problem can be viewed as a special case of the multicast network coding problem [34, 35]. Through this connection, we establish the optimal data access by invoking the well known multicast network coding result [34, 35] on an equivalent data delivery network. On the other hand, although this network coding connection is sufficient to establish the existence of such optimal codes for delayed parity generation, it does not provide an explicit and efficient code construction. Our main contribution of this work is an explicit and low complexity construction based on a strategic transformation on any MDS codes (such as the Reed-Solomon codes) [36]. The proposed code is partly inspired by the recently proposed generic transformation on data storage codes for optimal repair bandwidth and rebuilding access [37], however the proposed code is simpler than that in [37].

Readers familiar with regenerating codes may immediately recognize that regenerating codes [38, 39, 40, 41, 42, 43, 44, 45], particularly minimum storage regenerating (MSR) codes, can be used in the scenario of interest, by viewing the node regeneration process as the delayed parity generation process. In fact, this was indeed advocated in [46, 47, 48] as a solution to the second application discussed above. However, regenerating codes are in fact unnecessary and wasteful in the scenarios of our interest. More precisely, in regenerating codes, the requirement is that the system can regenerate *any* failed nodes (the number of which is less than a certain threshold) using the minimum amount of data traffic (or data access) from the surviving ones, however, in the problem that we are considering, the requirement is that the system can generate *a single fixed set* of new parity nodes with the minimum amount of data access from the existing ones. Somewhat surprisingly, this relaxation in the problem setting does not lead to a reduction in the minimum amount of data access, however it does lead to other benefits, such as a much reduced subpacketization factor and less coding computation.

Indeed, it was established in [44, 49] that regenerating codes with optimal repair traffic and disk IO require a subpacketization (the number of symbols in each node) at least equal to the number of parity nodes raised to the power of the number of data nodes divided by the number of parity node; in contrast, the construction we propose has a subpacketization equal to the number of parities. This reduction is extremely important in practice since firstly, this implies a much simpler code and lower computation burden, and secondly, the subpacketization factor dictates the minimum amount of data stored on each node, i.e., codes with a high subpacketization factor can only be used for large data files. Moreover, the code we propose in this work uses a small alphabet, in clear contrast to most known regenerating codes [38, 39, 40, 50, 41, 44, 42]. It should also be noted that the fundamental limits derived for regenerating codes in [38] do not directly apply in our problem due to the relaxed setting, and thus we need to derive new outer bounds for it.

Although the motivation is to improve the data write speed (data commit speed) in large scale erasure-coded systems, the problem turns out to be closely related to the so-called adaptivity (or scaling) of erasure-coded data storage systems [46, 47, 48]. The goal of such systems is to allow the adaptation of the number of parities in the erasure-coded systems when data has already been committed to the said system. One can show that the two problems are in fact equivalent in a sense, which we shall explain in the latter part of this paper. Interestingly, previous efforts [46, 47, 48] on the adaptivity in fact took the approach of directly applying regenerating codes, which, for the same reason as we have just explained, is rather unnecessary and wasteful. Thus, the codes proposed in this work can also be used to provide adaptivity or scaling in erasure-coded systems with a much lower subpacketization factor, and much lower computation complexity, than the solutions based on generating codes previously given in the literature.

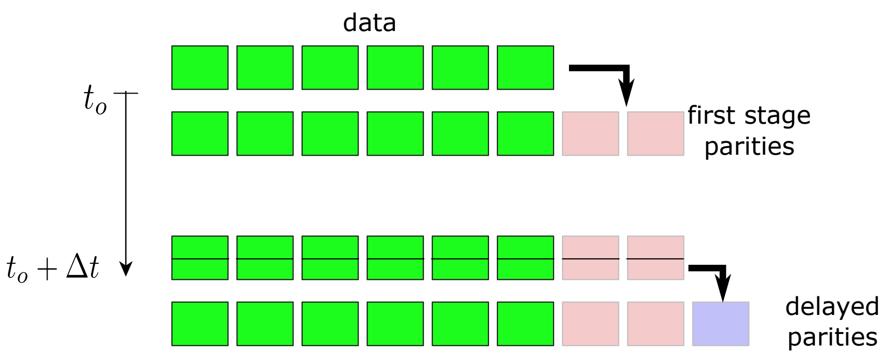

Figure 1.1: The first stage parities are generated from the raw data, while the delayed parities are generated by reading part of the data and the first stage parities.
reprinted with permission from [1]

This delayed parity generation (DPG) scheme could be summarized that the basic idea is to only generate and write a subset of the parity symbols together with the raw data, but delay the generation of the remaining parity symbols to a later time. These delayed parities are then generated and written, when high reliability is needed or when the system load is lighter, by accessing a subset of the information written in the first stage; see Fig. 1.1.

Since the time between initial data write and delayed parity generation is a design parameter, the loss of the reliability during that short time period can be controlled and is compensated by the potential gain in performance. This approach is particularly suitable for systems where computation resource is less constrained, but the network or IO bandwidth is constrained under workload fluctuation. Since CPUs have become more powerful over the years, we anticipate that bandwidth will continue to be the constraining factor in the overall data storage pipeline.

From a practical perspective, we consider two issues related to the application of delayed parity generation in distributed data storage systems. The first issue is to optimize the underlying codes for delayed parity generation. A generic code construction to facilitate delayed parity generation was proposed in [1] as a transformation on any existing storage code. However, it is not optimized for performance. There were significant efforts in the literature [29, 26, 2, 28, 27, 32, 31] on optimizing conventional erasure codes for better encoding performance. In order to leverage these existing techniques, we first provide a generalization of the delayed parity generation code, and then optimize it in the well-known bitmatrix-based framework; we refer to this approach as BB-DPG (bitmatrix-based DPG). The optimized codes help the system avoid being computation bounded. The second issue is to understand the impact of delayed parity generation in a system setting. For this purpose, we use a prototype implementation to verify whether this approach can indeed offer performance improvement. The prototype system is tested on a local network, as well as on virtual machine based testing platforms. In general, we observe that when the system bottleneck is the communication bandwidth, delayed parity generation can indeed provide performance improvement during the data write process.

2. BACKGROUND [1]

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

$$v = u \cdot G, \tag{2.1}$$

where

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

For scalar codes, the k raw data symbols are written as a row vector $\vec{d}$ of length k, each entry

[1] ©2020 ACM. Partly Reprinted, with permission, from T. Zhou and C. Tian. 2020. "Fast Erasure Coding for Data Storage: A Comprehensive Study of the Acceleration Techniques". ACM Trans. Storage 16, 1, Article 7 (March 2020), 24 pages. DOI:https://doi.org/10.1145/3375554

of which is in a certain finite field. The overall encoding process can be viewed as multiplying the vector $\vec{d}$ with a generator matrix G of dimension k-by-$(k+m)$, the entries of which are also in . When the left k-by-k submatrix of G is an identity matrix (such codes are referred to as *systematic codes* [36]), the product can be written as the concatenation of two vectors $[\vec{d}, \vec{p}]$, where the length m vector $\vec{p}$ is referred to as the parities. Mathematically, this can be expressed as

$$[\vec{d}, \vec{p}] = \vec{d} \cdot G = \vec{d} \cdot [I_k, P]. \tag{2.2}$$

In order to achieve the best data storage efficiency, the matrix G needs to be chosen such that the submatrix of any k columns of G is full rank. In other words, having any k out of the $k+m$ symbols in the vector $[\vec{d}, \vec{p}]$ allows recovery of the original data vector $\vec{d}$, through a matrix inversion operation. Thus, when the symbols in $[\vec{d}, \vec{p}]$ are distributed to different storage devices (nodes), loss of m such devices (nodes) will not cause any data loss. There are many different methods known to choose the matrix P; see e.g., [14, 51].

An erasure code can be defined as follows. With a given k storage nodes, additional m parity nodes will be generated. All these $n = k + m$ nodes have the probability to be failed. The storage system will be able to recognize the nodes failure and the term *erasure* is used to describe the this failure. An erasure code will define how the m parity nodes are generated and how to recover the data from non-erasure nodes in those $k+m$ nodes. This is usually written as $(k+m, m)$ or (n, m) erasure code.

The procedure of generating the m parity nodes is named *encoding* and the k nodes which contains original data are named *systematic nodes* whereas the procedure of recovering the original data from survived nodes if nodes failure happens is called *decoding*.

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 this invertibility, i.e.,

$$G = \begin{bmatrix} 1 & \cdots & 1 & \cdots & 1 \\ a_0 & \cdots & a_i & \cdots & a_{n-1} \\ a_0^2 & \cdots & a_i^2 & \cdots & a_{n-1}^2 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_0^{k-1} & \cdots & a_i^{k-1} & \cdots & a_{n-1}^{k-1} \end{bmatrix} \tag{2.3}$$

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., [11]) to obtain an equivalent generator matrix G'

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

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

$$v = u \cdot G' = (u_0, u_1, \cdots, u_{k-1}, p_0, p_1, \cdots, p_{m-1}), \tag{2.4}$$

where

$$(p_0, p_1, \cdots, p_{m-1}) = (u_0, u_1, \cdots, u_{k-1}) \cdot P. \tag{2.5}$$

The matrix P is sometimes also referred to as the coding distribution matrix [52].

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 invertibility 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 (CRS) codes [29].

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 at the i-th row and the j-th column 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 [29].

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 [2], 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 [29], and thus all the computations can be conducted using direct copy or binary XOR.

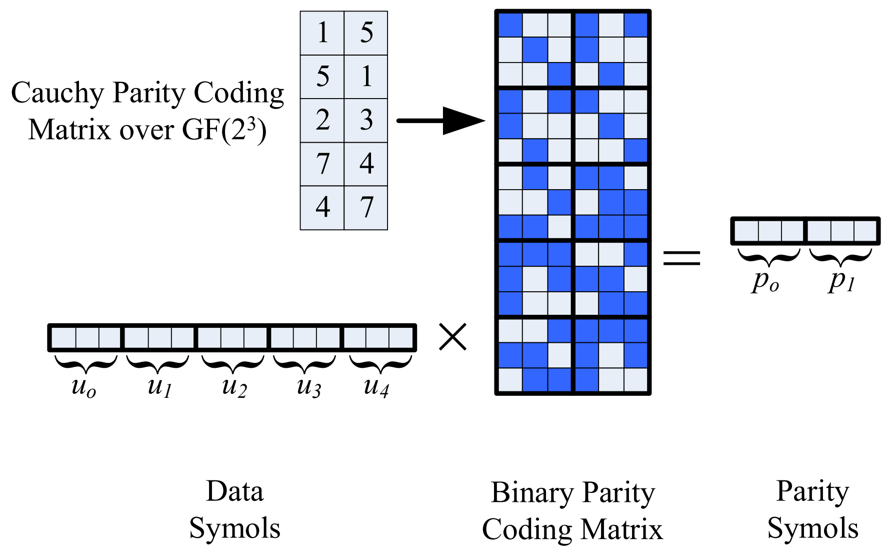
Cauchy Parity Coding Matrix over GF(2^3)
1 5
5 1
2 3
7 4
4 7
u_o u_1 u_2 u_3 u_4
p_o p_1
Data Symols
Binary Parity Coding Matrix
Parity Symols

Figure 2.1: 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. Adapted from [2]. reprinted with permission from [3]

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. The details can be found in [29], however, a simple example of bitmatrix encoding is shown in Figure 2.1, 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 procedure, if we use this bitmatrix representation to implement the MDS coding process. Choosing different $X = (x_1, \dots, x_k)$ and $Y = (y_1, \dots, y_m)$ 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 [52], 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, over that using the bitmatrices without applying such heuristics.

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 suitable for computation (e.g., to make some elements be the multiplicative unit 1), which is referred to as bitmatrix normalization [2]. 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.

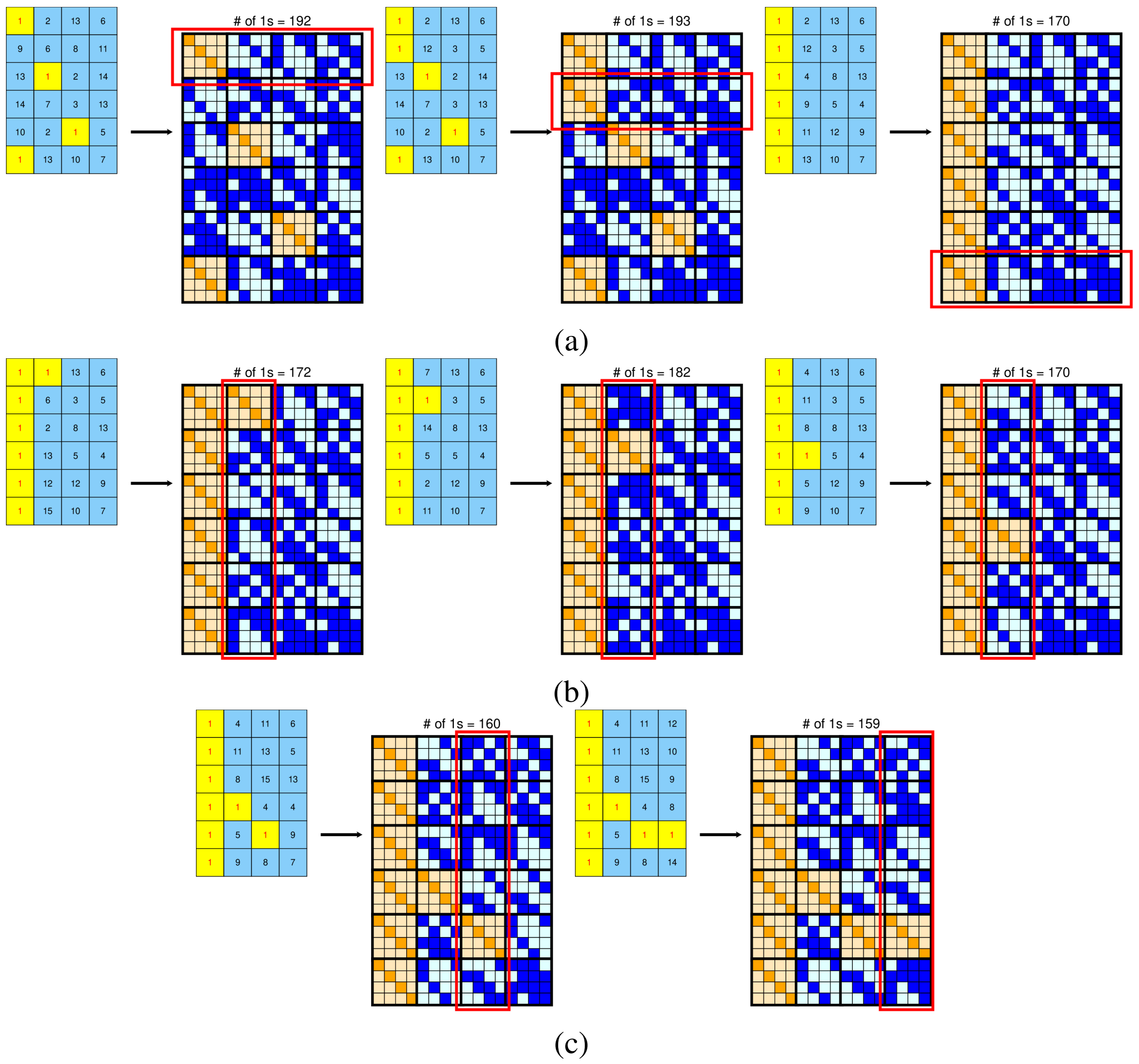
Figure 2.2: Bitmatrix normalization, (a) Normalize the first column to all 1's in finite field, as described in step 1; (b) Normalized the second column to keep the one with the minimum total number of bit 1's, as described in step 2; (c) Step 2 applied to all the rest columns

An example given by Plank et al. [2] 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. A visualization of this procedure is shown in Fig. 2.2.

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 [30], and this idea materialized as the smart scheduling component in the software package [2]. 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

$$p_0 = u_0 \oplus u_2 \oplus u_3 \oplus u_4, \quad p_1 = u_0 \oplus u_2 \oplus u_3 \oplus u_5. \tag{2.6}$$

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 also 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. [26] 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 [53] is utilized to implement these matching algorithms.

Generalizing the matching technique, a few more heuristic methods to reduce the number of XORs were investigated by Plank et al. [31]. However, these methods themselves can be extremely time-consuming (taking hours or even days), and the observed improvements in coding throughput appear marginal. Therefore, we do not further pursue these heuristic methods in this work.

A simple example could be find in Fig. 2.3. When we compute the three green parity bits, they all need the XOR of two message bits corresponding to the red bit 1's in the coding bitmatrix, we call it a *pair*. This matching technique can help find these *pairs*, pre-compute them and re-use them. To help reduce the total number of operations in a coding scheme.

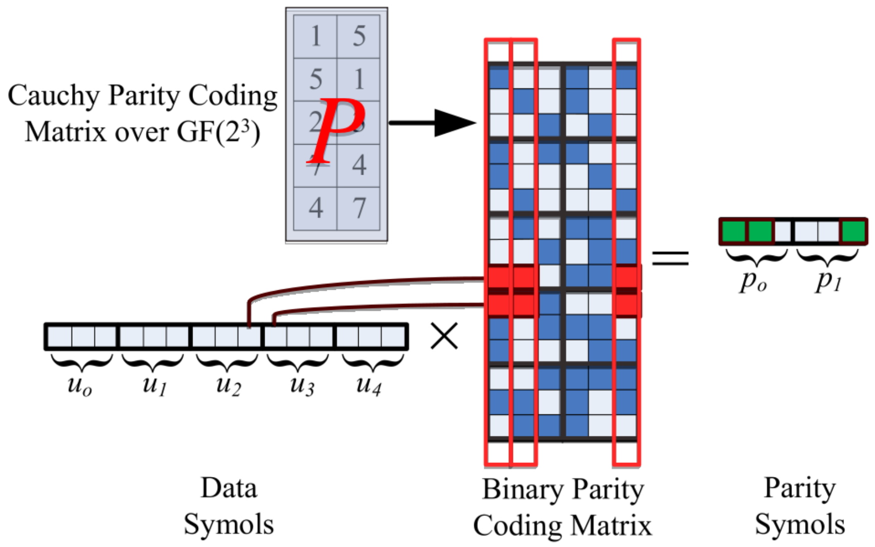
Cauchy Parity Coding Matrix over GF(2³)
1 5
5 1
2 3
7 4
4 7
P
u_0 u_1 u_2 u_3 u_4
Data Symols
Binary Parity Coding Matrix
p_0 p_1
Parity Symols

Figure 2.3: *Matching pair* in coding scheme

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 throughput. 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 $GF(2^{32})$ [28]. This was accomplished through specially designed table lookup method for each field size, and then invoking the 128-bit SSE vectorization instruction set for Intel®, AMD, and ARM CPUs. More recently, 256-bit AVX2 vectorization instructions and 512-bit AVX-512 vectorization instructions are becoming more common in newer generations of CPUs. The popular ISA-L library [33] developed by Intel® also adopted the approach of vectorizing finite field operations directly, which includes implementation for the finite

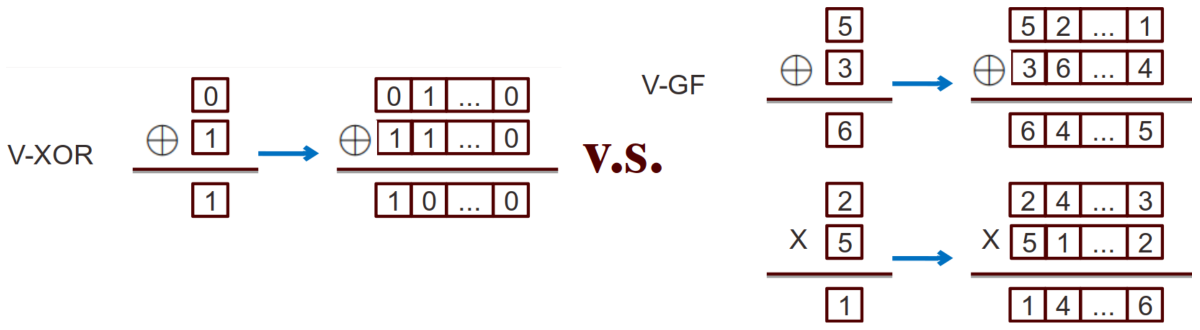

Figure 2.4: Vectorization

field $GF(2^8)$ but for more diverse vectorization widths (128-bit, 256-bit, or 512-bit).

It should be clear at this point why the finite fields $GF(2^8)$, $GF(2^{16})$, and $GF(2^{32})$ were chosen to be vectorized: these finite fields are better aligned with the number of bits in a computer byte. In contrast, when the bitmatrix representation is used, since the operation is essentially on one of the bits in the binary extension field, there does not exist any particular constraint or difference in the implementation for different field sizes.

We note that the open source libraries evaluated in [54] did not use vectorization and in fact advanced SIMD instructions were not widely available at the time. However, the new version of Jerasure [28] and ISA-L libraries are indeed based on this technique. As a consequence, the performance tests we conducted in this work will need to take into considerations this new development in the field.

In Fig. 2.4, we demonstrate two types of vectorization, XOR-based vectorization and GF-based vectorization. For XOR-based vectorization, it focus on compute the bit-wise XOR of two groups of input data bits. The group length could be arbitrary chosen as long as it's a multiple of CPU vectorization register width (128-bit, 256-bit or 512-bit). GF-based vectorization, on the other hand, can perform symbol-wise addition and multiplication of two groups of input data symbols. Each symbol is one element in finite field, which is typically $GF(2^8)$, $GF(2^{16})$, and $GF(2^{32})$. XOR-based vectorization is deployed in our proposed code and GF-based vectorization will be included when comparing the coding throughput.

2.3.2 Reducing Cache Misses

The order of computing 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 were given by Luo et al. [27]. 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%$\sim$36%.

This strategy could help increasing data locality, in another word, increasing cache hit rate, it has two sequence of XOR operations: 1) parities first; 2) data first.

1. Parities first strategy will calculate each parity first, it reads all data necessary for calculating p_1, then calculate p_1. Repeat this procedure to all parities. This strategy will potentially make duplicate readings of the same data memory address, which may cause cache replacement and cache miss penalty.

2. Data first strategy will read d_1 first, then update all parities which demands d_1. Repeat this procedure to all data. This strategy could help increasing data locality, in another word, increasing cache hit rate.

This reordering of XOR operations could reduce data read as well as cache miss rate. It's results shows up to 36% improvement by just reordering the sequence of operations.

In our implementation, we reorder the schedule by the appearance of each data to achieve this spatial locality. Theoretically, for XOR t symbols, only $t-1$ XOR operations is needed. In implementation, however, we need to set the destination memory to the first source symbol. This happens in both calculation of intermediate results in the matching phase as well as parities. In our application, we reorder the schedule by the appearance of each data, then modify the operation of the first appearance of each destination from XOR to memory copy (memcpy in standard C library). In our specified test environment, memcpy is around twice faster than 128-bit vectorized XOR operations, and $\sim 30\%$ faster than 256-bit vectorized XOR operations..

3. ACCELERATING TECHNIQUES [1]

3.1 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.

3.1.1 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.

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 3.1, 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 XOR operations in the schedule, while for the latter two, the improvement is measured in terms of the encoding throughput increase. Multiple tests are performed for

[1] ©2020 ACM. Partly Reprinted, with permission, from T. Zhou and C. Tian. 2020. "Fast Erasure Coding for Data Storage: A Comprehensive Study of the Acceleration Techniques". ACM Trans. Storage 16, 1, Article 7 (March 2020), 24 pages. DOI:https://doi.org/10.1145/3375554

Table 3.1: Performance improvements by individual techniques, reprinted with permission from [3]

(n,k,w)	# of XORs: baseline	# of XORs: reduction				Throughput increase	
		BN	SS	UM	WM	S-CO	V-XOR
(7,5,3)	45	31.11%	11.11%	28.89%	28.89%	0.25%	43.87%
(7,5,4)	79	34.18%	24.05%	32.91%	31.65%	0.59%	49.98%
(7,5,8)	267	49.06%	26.59%	34.46%	34.08%	0.09%	66.13%
(8,6,3)	60	31.67%	13.33%	31.67%	31.67%	0.09%	48.58%
(8,6,4)	104	42.31%	17.31%	31.73%	31.73%	0.14%	53.86%
(8,6,8)	362	53.31%	33.70%	34.53%	34.81%	0.05%	72.72%
(9,6,4)	152	32.89%	19.74%	32.89%	32.89%	0.10%	57.02%
(9,6,8)	549	44.63%	29.14%	38.25%	38.25%	0.45%	76.66%
(10,6,4)	200	27.50%	22.00%	36.00%	36.00%	0.18%	62.25%
(10,6,8)	736	40.90%	28.80%	38.59%	38.59%	0.20%	79.70%
(10,7,4)	168	26.79%	13.10%	31.55%	32.14%	0.18%	59.35%
(10,7,8)	633	49.13%	24.49%	37.60%	36.81%	0.51%	77.28%
(11,8,4)	192	27.60%	20.83%	33.85%	32.29%	0.00%	64.85%
(11,8,8)	771	41.37%	22.05%	38.00%	37.61%	0.31%	80.12%
(12,8,4)	256	23.44%	23.44%	35.94%	35.94%	0.28%	68.04%
(12,8,8)	1028	36.38%	24.81%	39.79%	39.79%	0.30%	81.80%
(14,10,4)	324	19.14%	16.36%	35.49%	35.80%	0.06%	71.66%
(14,10,5)	488	23.98%	20.29%	36.27%	36.07%	0.39%	75.80%
(14,10,6)	728	28.85%	20.88%	37.77%	37.77%	0.16%	79.52%
(14,10,7)	890	26.9%	11.46%	37.53%	37.53%	0.33%	80.44%
(14,10,8)	1234	32.58%	17.26%	38.74%	38.74%	0.49%	82.59%
(16,10,4)	496	18.95%	21.77%	37.70%	37.70%	0.41%	76.90%
(16,10,8)	1920	30.16%	21.98%	40.89%	40.89%	0.00%	86.54%
Average over all tested cases		35.36%	20.22%	35.36%	35.34%	0.31%	70.64%

each parameter, and we report the average over these runs. In the last row of Table 3.1, the average encoding throughput over all the tested parameters is included.

All tests in this section are conducted on a workstation with an AMD Ryzen 1700X CPU running at 3.4GHz, 16GB DDR4 memory, which runs the Arch Linux operating system and the compiler is GCC 8.3.0. During compilation, -O3 optimization 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 in terms of XOR operations. 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 $\sim 20\%$ 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. We emphasize again that here the gain is measured in terms of the reduction in the number of XORs, however in real systems, the precise throughput gain is more complex with several other factors such as *packet_size* and cache management also impacting the performance. It should also be noted that these procedures are usually not performed online during data encoding, but done in an offline fashion during the design stage.

Between the latter two techniques, S-CO can provide a neglegible gain of $\sim$0.3%, while V-XOR is able to improve the coding throughput by $\sim$70%. In fact, among all the techniques, V-XOR appears to be able to provide the most significant performance improvement. The improvement by S-CO observed in our work is considerably less significant compared to the 23-36% improvement reported by Luo et al. [27], which we suspect is due to the improvement in cache size and cache prediction algorithm in modern CPUs. The packet size parameter (i.e., how large a data block is being processed together each time) in fact affects this value significantly. This impact by the

Table 3.2: Combination of individual strategies, reprinted with permission from [3]

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

packet size parameter will be revisited in more details later on in Section 3.3.5. The improvement by V-XOR requires some further explanation. Here the XOR operation in the baseline reference procedure is implemented using "long int" data type pointers, and thus each XOR operation can be viewed as a mini vectorized operation on 64 bits. Some readers may realize that when the flag -O3 is enabled in GCC, the compiler will attempt to auto-vectorize computations whenever possible. However, in our implementation and platform, it turns out that this auto-vectorization does not succeed, when no further intervention is adopted. The relation between auto-vectorization and explicit vectorization will be revisited in Section 3.3.4.

The performance of directly vectorizing finite field operations [28] 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 vectorizing 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.

3.1.2 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 3.1.

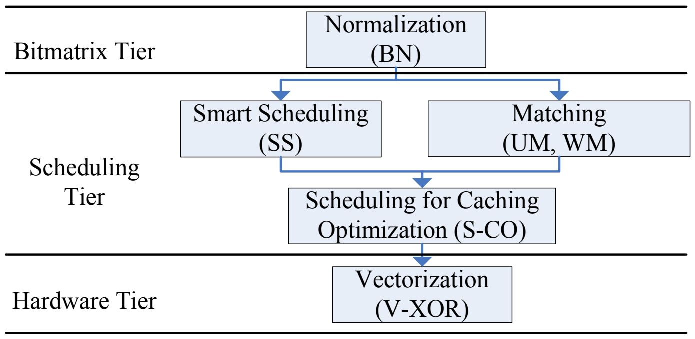

Figure 3.1: Operation tiers of the individual techniques, reprinted with permission from [3]

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 to 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 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, its inclusion 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.

3.2 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 [52]), 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.

3.2.1 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 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.

3.2.1.1 Optimization Algorithm Details

Note that for the strategies where normalization, smart scheduling, or matching algorithms are used, these cost functions are computed after these procedures are conducted. Thus the mapping from the (X, Y) vector to the cost functions can be rather complicated, and there does not exist an explicit expression for this mapping. As detailed in the sequel, simulated anneal and genetic algorithms are used to find better (though not necessarily optimal) (X, Y) assignments.

For easier understand the results, we normalized the encoding cost by the cost that no optimization applied, which is the "Natural" data in the figure. These test results show that normalization should always be applied, matching outperforms smart scheduling, weighted matching works better than unweighted matching in more cases. Among all these combinations, performing normalization then unweighted matching has the best performance in most cases.

The result shows that we can achieve $\sim 5\%$ to $\sim 25\%$ reduction of encoding cost if searching mechanism applied.

Besides k, m, w in Reed-Solomon code, we introduced a new parameter set S. Then we formalize the searching mechanism to:

For a given k, m, w, S input, the searching algorithm starts from a random sequence X, Y, where $len(X) = k, len(Y) = m, X, Y \in GF(2^w)$, Then we perform algorithm specified iteration, trace the minimum encoding cost and stop if the minimum value hasn't changed for S consecutive iterations.

In simulated annealing algorithm, parameter $Rate_{accept}$ indicates the initial acceptance rate of a successor which have worse value than current minimum. This rate will decrease as the time grows. We choose the annealing function $e^{-\Delta T}$, where Δ is the annealing factor.

Another searching algorithm is based on the hypothesis that some number or number combination in $GF(2^w)$ leads to less encoding cost over others. A genetic algorithm will be fit in this situation. It has following parameters.

1. *person*, we define a person contains a vector of distinct symbols on $GF(2^w)$ with length n, where the first k symbols is used as X array and rest m symbols as Y array as defined in Cauchy Matrix.

2. $Rate_{select}$, for each iteration, the top $Rate_{select}$ of population will be selected as potential parents. When the population reach the maximum population, the lowest $Rate_{select}$ of pop-

ulation will be randomly weeded out until the population is less or equal to the maximum population.

3. $Rate_{crossover}$, for any pair of parents, they have a rate of $Rate_{crossover}$ to generate their child.

4. $Rate_{mutation}$, in each iteration, every people has a chance of $Rate_{mutation}$ to change a random number in the data array to another unused number of that people.

5. *max_ population*, the maximum population.

In crossover phase, we're using two methods to generate there child.

1. Random crossover: A set of symbols on $GF(2^w)$ are collected which each number occurs at least once in one of the parent, then we pick random $(k+m)$ symbols in that set as the data array of child, A small test shows using this method. The probability that the child's performance is better than at least one of its parent is $\sim 2\%$.

2. Common elements first crossover: based on our hypothesis, we're assuming that some number is better than others. In this method we will consider the numbers occur in both parents first as the element of child. Using this method, we observed that $\sim 25\%$ of children has better performance than at least one of their 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 traits in the children may produce even better assignments.

3.2.1.2 Parameter Effect

For better understanding these optimization algorithms, some quick tests about different value of algorithm parameters is performed. In each test, we modify only one of the parameters and fix all others. The results are shown in Tab. 3.3 and Tab. 3.4. We pick 10 total different values for each parameter in a reasonable range and record the obtained coding schedule length. All the numbers in the results are the percentage of the worst case when changing one parameter over the overall best result, in term of length of schedule.

Table 3.3: Parameter effect of Simulated Annealing Algorithm

(n,k,w)	(9,6,4)	(12,8,4)	(16,10,4)
annealing factor	88.32%	93.48%	96.50%
S	88.25%	93.51%	96.66%
$Rate_{accept}$	88.18%	93.59%	96.60%

Table 3.4: Parameter effect of Genetic Algorithm

(n,k,w)	(9,6,4)	(12,8,4)	(16,10,4)
$Rate_{select}$	88.10%	93.40%	97.37%
$Rate_{crossover}$	88.10%	93.19%	97.62%
$Rate_{mutation}$	88.10%	92.97%	97.84%
max_population	88.10%	93.01%	97.55%
S	88.10 %	93.24%	97.69 %

From observing the speed of encoding, changing the parameters is not showing much difference of performance.

3.2.1.3 Bitmatrix Optimization Results

In Table 3.5, 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 for conciseness. 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 3.5, that both SA and GA can provide significant improvement on the total number of operations by finding good bitmatrices. In the ranges of tests, we typically observe improvements of between 5%-25%, in terms of the total number of XOR and copy operations. In Figure 3.2, we further plot the amounts of cost reduction for different (n, k, w) parameters, from the baseline approach of

Table 3.5: 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, reprinted with permission from [3]

(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**

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.

In these results, we observed that Genetic algorithm using weighted operations as cost function could outperform other combinations 1%$\sim$10%. Depends on this observation, we use the output array X and Y of Genetic algorithms and weighted encoding cost for further test.

3.2.2 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 3.3, the cost function values of different (i, j)-strategies are shown, under various (n, k, w) parameters. Here again we have chosen a subset of representative test results, and omitted others for the sake of conciseness. 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

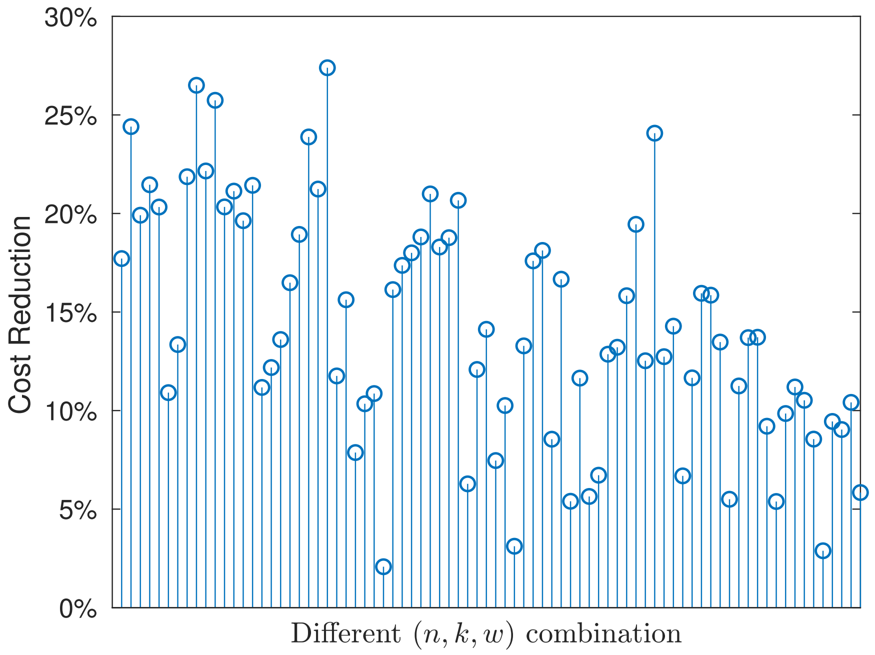

Figure 3.2: Cost reductions obtained by the genetic algorithm for different (n, k, w) parameters (sorted by n).

between themselves.

3.2.3 Refining the Cost Function

We have so far used the total number of XOR 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 weights 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 times 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 function is evaluated and shown in Table 3.6 by using the genetic algorithm to find the optimized bitmatrices. The resulting bitmatrices obtained under

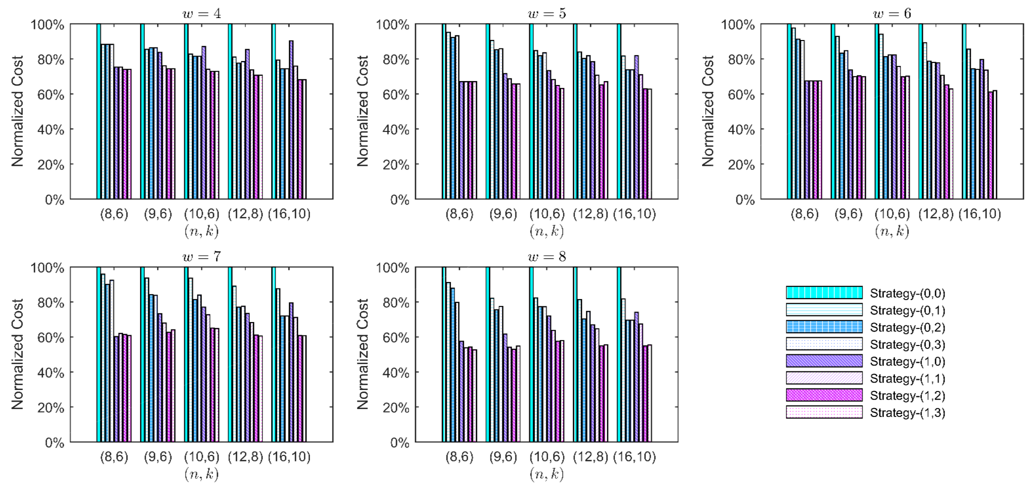

Figure 3.3: Total number of XOR and copying operations for all (i, j)-strategies with optimized bitmatrices, reprinted with permission from [3]

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 modest, ranging from $0\% - 10\%$, and occasionally it does cause a minor performance degradation than the cost function $c_{i,j}(X, Y)$.

3.3 The Proposed Design and Coding Procedure Details

From the previous discussion, the proposed bitmatrix design procedure is quite clear: perform a suitable optimization procedure (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 design then naturally utilizes this selected (X, Y) vector to produce the corresponding bitmatrix by normalization, then generate the computation schedule from the resulting bitmatrix using the selected matching algorithm and optionally following the cache-friendly order. The computation schedule succinctly encapsulates the sequence of copying and XOR operations, and can be generated offline. Correspondingly, the proposed coding procedure follows this computation schedule during run-time, but performs the vectorized copying and XOR operations using the necessary CPU instructions.

Table 3.6: Encoding throughput (GB/s) with bitmatrices optimized under different cost functions, reprinted with permission from [3]

(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**

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

3.3.1 Integrating XOR Matching

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. Though in our test, the benefit in coding throughput following this order is negligible, in the proposed procedure, the same idea can be used to better organize the computation schedule.

The schedules in [27] contain only data to parity operations

$$u_i \to p_j,$$

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

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

We can 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.,

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

In this order each data bit u_i will be read exactly once, and the computation is indeed done correctly.

3.3.2 Vectorizing XOR Operations

Each step in the computation schedule is either a copying operation, or an XOR operation. The smallest such computation unit is a byte (8bits) in computer systems, and similarly a 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 vectorization instructions we used in this work are included in the Intel® Intrinsics instruction set, which has also been adopted in AMD CPUs; in ARM based CPUs, the equivalent instructions set is the NEON Intrinsics instruction set.

The C code snippet to perform the vectorized XOR operation on x86/x64 platforms 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(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`

Similarly, on an ARM-based platform, the only additional difference is the I/O between the memory and NEON registers needs to be called explicitly. More precisely, the C code to perform the XOR operation on an ARM CPU is as follows:

```
#include <arm_neon.h>
void fast_xor( char *r1,   /* region 1 */
               char *r2,   /* region 2 */
               char *r3,   /* r3 = r1 ^ r2 */
               int size)   /* bytes of region */
{
    int j;
    uint8x16_t b1, b2, b3;
    int vec_width = 16;
    int loops = size / vec_width;
    for(j = 0;j<loops;j++)
    {
        b1 = vld1q_u8(r1+j*vec_width);
        b2 = vld1q_u8(r2+j*vec_width);
        b3 = veorq_u8(b1,b2);
        vst1q_u8(r3+j*vec_width, b3);
    }
}
```

On our AMD CPU setup, performing the XOR operation on the same amount of data with 64-bit width (long format) appears to be 30% slower than `_mm_xor_si128`, and 50% slower than `_mm256_xor_si256`. Clearly, different CPUs may have different characteristics, and the precise amount of improvements may vary, however in general, the wider the vectorization, the high the throughput can be expected.

3.3.3 Encoding Performance Evaluation with Fixed Packetsize

In the early stage of this research, a typical packetsize (16 KB) has been chosen to perform throughput evaluation. Some preliminary results is listed here even further discussion about packetsize in Sec. 3.3.5.

From previous results, we observed the following facts:

1. Genetic algorithm will give out better search result than simulated annealing, changing the parameters of these two algorithms will not affect much of the search result. Meanwhile, searching is still necessary for better performance;

2. Using normalization is always better than not using;

3. Matching method outperform smart scheduling in most cases;

4. Unweighted matching and weighted matching will not make much difference.

5. Schedule reordering and vectorization should always be applied.

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 [23], RDP code [24], Linux Raid-6 [30], STAR code [25], and Quantcast-QFS[12] 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 3.3.2 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 includes several efficient Cauchy Reed-Solomon code implementations based on bitmatrices (XOR-based CRS) [2, 28, 27], and finite field vectorized Reed-Solomon code (GF-based RS code) [28]; the source code for them can be found in the Jerasure library 2.0 [28] publicly available online, which is used in our comparison. The implementation in Jerasure library 2.0 is based on vectorizing

Table 3.7: Encoding throughput (GB/s) for methods that allow general (n, k) parameters and $w = 8$ with fixed packetsize, reprinted with permission from [4]

(n, k)	Proposed	Vectorized XOR-based CRS code	Vectorized GF-based RS code [28]
(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

Table 3.8: Encoding throughputs (GB/s) with fixed packetsize: Three parities, reprinted with permission from [4]

(n, k, w)	Proposed	Vectorized XOR-based CRS code	STAR code [25]	Quancast QFS [12]
(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

(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 [2] can be adapted to utilize with XOR-level vectorization, however, it would not include the UM (or WM) component and the refinement of the cost function discussed in Section 3.2.3.

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 3.7, 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 [28] 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 XOR-SS technique in these cases in fact provides a slighter better scheduling than XOR-WM. When m is larger than 2,

Table 3.9: Encoding throughputs (GB/s) with fixed packetsize: Two parities, reprinted with permission from [4]

(n,k,w)	Proposed	Vectorized XOR-based CRS code	Vectorized Raid-6	EVEN ODD [23]	RDP [24]
(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**

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 3.8, 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 3.9, 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.

In Table 3.10, 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,

Table 3.10: Encoding throughput improvements over references with fixed packetsize, reprinted with permission from [4]

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 [28]	99.82%
Vectorized XOR-based CRS code	14.98%
Three Parities Codes	
STAR [25]	5.59%
Quancast-QFS [12]	21.68%
Two Parities Codes	
Raid-6 w/o vectorization	206.88%
Vectorized Raid-6	142.07%
RDP [24]	5.85%
EVENODD [23]	8.79%

however for individual (n, k, w) parameter, the performance may vary as indicated by the previous three tables.

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 useful 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.

Table 3.11-3.13 give the decoding throughput corresponding to table 3.7 to 3.9. Our decoding test covers all the worst case data losing, which lost m systematic nodes. These cases are the most computational consuming cases.

Table 3.11: Decoding throughput (GB/s) for methods that allow general (n, k) parameters and $w = 8$ with fixed packetsize, reprinted with permission from [4]

(n, k)	Proposed	Vectorized XOR-based CRS code	Vectorized GF-based RS code[28]
(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 3.12: Decoding throughputs (GB/s) with fixed packetsize: Three parities, reprinted with permission from [4]

(n,k,w)	Proposed	Vectorized XOR-based CRS code	STAR code [25]	Quancast QFS [12]
(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

As seen in Table 3.11, 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 3.12 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 3.9, the decoding throughput of proposed approach is usually lower than EVEN-ODD and RDP codes.

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

3.3.4 Auto-vectorization vs. Explicit Vectorization

Most of the newer compilers have the capability of recognizing the opportunity of vectorizing certain computation in a loop. In GCC, the compiler option -O3 would in fact enables auto-vectorization, even when the source code itself does not explicitly make calls using the SSE (or AVX) instructions. However, we found that the compiler treats the XOR operation and finite field multiplication very differently in this regard. For the former, the compiler will recognize

Table 3.13: Decoding throughputs (GB/s) with fixed packetsize: Two parities, reprinted with permission from [4]

(n,k,w)	Proposed	Vectorized XOR-based CRS code	Vectorized Raid-6	EVEN ODD [23]	RDP [24]
(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**

a loop to perform XOR operations as potentially vectorizable, whereas for the latter, it will not. This is hardly surprising since XOR is a simple computation, whereas finite field multiplication is a much more involved sequence of computations. Moreover, even if the compiler could auto-vectorize finite field multiplications, it is likely to perform much worse than the specially designed vectorized version. In fact, this is part of the motivation to develop Jerasure 2.0 library and the ISA-L library in the first place.

Although the auto-vectorized XOR operations can be much more efficient than the non-vectorized version, this approach is not without its pitfalls. We found that in many cases, the compiler can produce the vectorized binary, which however is bypassed during runtime when certain conditions are not satisfied. The reason why this bypass happens is that in the *region_xor* routine, we're operating two pointers. The program bypass the vectorized binary of these pointers because of potential address overlap. For such cases, the directive `#pragma ivdep` can be used to force the runtime ignore any possible address overlap and avoid such runtime bypass. This approach of enforcing auto-vectorization, on one hand, does allow less platform dependent implementation without explicit calls to the SSE (or AVX) instructions. On the other hand, however, the binary such produced is usually 10%-20% slower than the explicitly vectorized version in our experience,

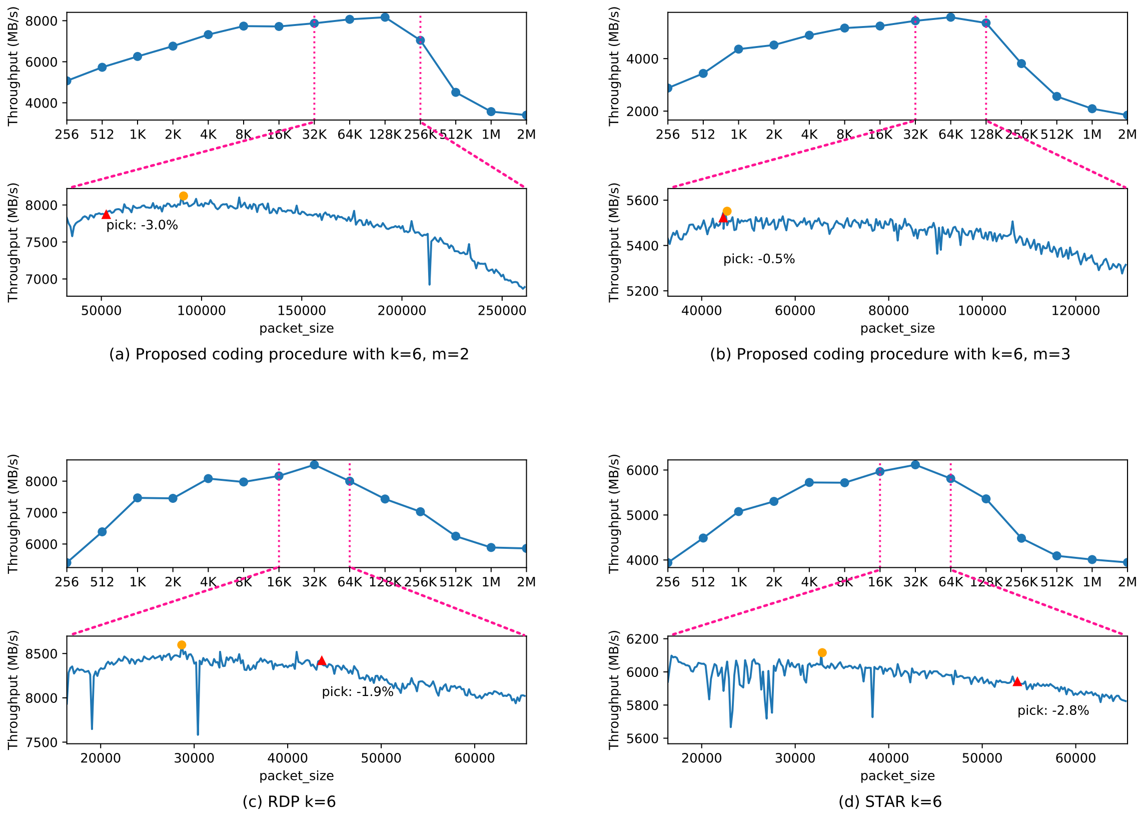

Figure 3.4: Encoding through v.s. *packet_size*, $k = 6, m = 2, 3$, for the proposed code, RDP and STAR, reprinted with permission from [3]

possibly due to other overhead associated with auto-vectorization.

3.3.5 Cache Size and Packet Size

In practice, a chunk of data is loaded into the CPU and processed as a group of bits; this chunk (for each u_i) is sometimes referred as a *packet*. We use *packet_size* to indicate the size of the packet, which is always a multiple of the minimum computing unit (32 bits, 64 bits, 128 bits, 256 bits, or 512 bits depending on the platform and vectorization capability): e.g., it can be chosen as 4K, 16K, or 64K bytes. The choice of *packet_size* will affect the throughput significantly, which was also observed by Luo et al. [27] and Plank et al. [54]. However, there does not exist a clear guideline in the literature on how to determine this parameter in a general manner. Through extensive testing, we observe that larger packet sizes usually increase the overall coding throughput, however, only when the data, intermediate results, and eventual computation results can be accommodated in the CPU cache of size L bits. For a computation schedule with t intermediate XOR bits, the packet size *packet_size* can be chosen to be the largest multiple of the number of bits in each computing unit, but satisfying the relation

$$\textit{packet\_size} * (w * n + t) \leq L. \tag{3.1}$$

Figure 3.4 shows the relationship between *packet_size* and the encoding throughput, for the proposed coding procedure, RDP code, and STAR code, respectively. Here the red triangle is used to indicate the choice of *packet_size* using the strategy given above, whereas the yellow circle indicates the optimal choice. It can be seen that the strategy given above indeed provides a good guideline, usually only incurring a very small fraction of loss comparing to the optimal choice. Although here we provide only a few samples on the AMD platform, similar phenomenon were also observed on other CPUs. For the performance evaluation in the latter part of the paper, *packet_size* is chosen using this strategy.

3.3.6 Decoding Procedure

As discussed by Plank [55], decoding erasure codes can be viewed as another encoding step, after the inverse of the submatrix of the generator matrix G is properly computed. In this work, we adopt the same strategy for decoding. However, because the bitmatrix is induced by the inverse of the generator submatrix, it is not subject to optimization. Nevertheless, the techniques of SS, WM, and UM can still be applied to generate an optimized computation schedule.

A complication arises when the *packet_size* is chosen as in (3.1), which can optimize the encoding throughput. The induced decoding bitmatrix will usually produce a computation schedule with more intermediate results, which implies that during decoding the cache cannot accommodate all the data, the intermediate results, and the eventual computation results. A simple alternative strategy is to use SS, instead of UM or WM. Because of the reduction of the intermediate results, the cache resource will be less constrained, and thus the decoding throughput improves considerably. We adopt this strategy for the decoding procedure in this work. It should be noted that this alternative decoding strategy is more effective with the suggested choice of *packet_size*, and for cache-constrained CPUs. On CPUs with a larger cache, this decoding strategy in fact may induce a small amount of performance loss, compared to that based on strategy UM or WM. Another possible alternative, if the storage system allows, is to let the decoding use a different *packet_size* than that in the encoding, which implies that the data will be written in and read out from the storage devices in different sized units that may not fully align. This approach of using a different *packet_size* in decoding does not perform as well as the proposed approach in most cases, because 1) a small *packet_size* tends to penalize throughput, 2) additional processing is most likely required due to the misaligned data, and 3) SS has a slight cache management advantage without the need to handle intermediate results.

3.4 Performance Evaluation

3.4.1 Reference Approaches and System Setups

In this section we will provide comprehensive encoding throughput evaluation between the proposed approach and several well-known efficient erasure coding methods in the literature, as well as the erasure array codes specifically designed for high throughput. The latter class includes EVENODD code [23], RDP code [24], Linux Raid-6 [30], STAR code [25], and Quantcast-QFS[12] 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. Open source implementations for some codes are not available, therefore we implemented these coding procedures in house, with and without vectorization, to use in our comparison. The former class includes several efficient Cauchy Reed-Solomon code implementations based on bitmatrices (XOR-based CRS) [2, 28, 27], finite field vectorized Reed-Solomon code (GF-based RS code) [28], and the popular ISA-L library by Intel® [33]; the source code for them can be found in the Jerasure library 2.0 [28] and the ISA-L 2.27 [33] publicly available online, which were 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 $GF(2^{32})$; ISA-L automatically chooses the widest vectorization possible on the platform but only allows $GF(2^8)$. The Cauchy Reed-Solomon code implementation in Jerasure library 1.2 [2] can be adapted to utilize XOR-level vectorization, however, it would not include the UM (or WM) component and the refinement of the cost function discussed in Section 3.2.3. In order to make the comparison fair (since the Jerasure 2.0 library only implements 128-bit vectorization), we provide results for both 128-bit vectorizition and the widest possible vectorization on the corresponding platform. These are the same setup as in [3].

We shall report single-core single-thread encoding and decoding throughput results on four different hardware platforms, which are shown in Table 3.14. These hardware platforms include Intel, AMD and ARM CPUs, with some being desktop CPUs and some being server CPUs. Various compilers and compiler settings have been tested, including GCC, “modern C compiler” CLANG, Intel C compiler ICC, and AMD C compiler AOCC. The throughput results do not show significant differences. In the sequel, all the performance results are reported for GCC 8.3.0 with -O3 optimization (i.e., auto-vectorization is enabled at compiling but bypassed during runtime), as well as the relevant compiler flags for explicit vectorization when necessary. Under the same (n, k) parameter, the encoding and decoding throughput decreases as w increases. Therefore, in the performance evaluation, w is chosen in the proposed coding procedure to be the minimum value that satisfies $2^w \geq n$, except when noted otherwise.

Table 3.14: Test hardware and software platforms, reprinted with permission from [3]

name	CPU	SIMD Support	Clock	L2 cache size	L3 cache size	MEM	OS
i7	Intel i7-4790	AVX2 (256 bit)	3.6 GHz	4×256 KB	8 MB	16 GB	Arch
amd	AMD Ryzen 1700X	AVX2 (256 bit)	3.4 GHz	8×512 KB	16 MB	16 GB	Arch
xeon	Intel Xeon Platinum 8163 (VPS)	AVX-512 (512 bit)	2.5 GHz	24×1 MB	33 MB	4 GB	Ubuntu
arm	AWS Graviton (VPS)	ARM-NEON (128 bit)	2.3 GHz	2MB	-	2 GB	Ubuntu

The various codes and coding procedures are abbreviated as follows in the sequel.

- **Jo-CRS**: the proposed coding procedure (**J**oint-**o**ptimized **C**auchy **R**eed-**S**olomon code), with optimized bitmatrix and XOR-based vectorizaion;
- **CRS**: similar to the proposed procedure with XOR-based vectorization but without UM or WM or the cost function refinement, adapted from Jerasure 2.0 [28];
- **ISA-L**: the ISA-L library [33] using $GF(2^8)$, version 2.27;
- **Jo-CRS-128**: the proposed coding procedure with only 128-bit vectorization;
- **JE-128**: Reed-Solomon codes with finite field based vectorization in Jerasure 2.0 [28], with 128-bit SSE vectorization and using $GF(2^8)$;
- **STAR**: Star code[25], with XOR-based vectorization
- **QFS**: Quantcast-QFS code[12], with XOR-based vectorization
- **Raid6**: Raid-6 code[30], with XOR-based vectorization
- **EO**: EvenOdd code[23], with XOR-based vectorization
- **RDP**: RDP code[24], with XOR-based vectorization

3.4.2 Encoding Performances

In the results reported in Table 3.15 to 3.24, the parameters k varies from 5 to 10, m from 2 to 6, and w from 3 to 8. The comparison is first presented in three groups.

- In Table 3.15 - 3.18, the proposed approach, with both 128-bit vectorization and wider vectorization, is compared with CRS, ISA-L, and JE-128 on different platforms. All these approaches are applicable for general (n, k) coding parameters. JE-128 is implemented only for 128-bit (SSE) vectorization, and for fairness a version of the proposed approach is implemented with only 128-bit vectorization. ISA-L mainly targets x86 (x64) architecture, and cannot efficiently utilize vectorization on ARM-based CPUs. As a consequence, its performance on this platform suffers significantly, and we do not include it in the comparison. Due

Table 3.15: Encoding throughput (GB/s) for methods that allow general (n, k) parameters (i7), reprinted with permission from [3]

(n, k)	Jo-CRS	CRS	ISA-L	Jo-CRS -128	JE-128
(7,5)	6.94 (6.64)	**7.02**	4.45	6.60 (6.06)	2.14
(8,6)	6.98 (6.15)	**7.94**	4.25	6.43 (5.54)	1.71
(9,7)	7.20 (6.25)	**7.92**	4.56	6.52 (5.49)	2.26
(10,8)	7.13 (5.89)	**7.98**	4.79	6.49 (5.17)	2.28
(12,10)	7.12 (6.03)	**7.92**	6.08	6.34 (5.22)	2.36
(8,5)	**5.37** (4.03)	5.23	3.13	4.87 (3.48)	1.44
(9,6)	5.52 (3.97)	**6.06**	3.06	4.85 (3.39)	1.49
(10,7)	5.31 (4.01)	**5.99**	3.33	4.75 (3.39)	1.53
(11,8)	5.59 (4.07)	**6.08**	3.68	4.73 (3.39)	1.52
(13,10)	5.49 (4.01)	**5.96**	4.15	4.66 (3.31)	1.57
(10,6)	4.37 (2.96)	**4.83**	2.48	3.81 (2.47)	1.13
(11,7)	4.36 (2.94)	**4.86**	2.86	3.79 (2.37)	1.15
(12,8)	4.34 (3.04)	**4.94**	2.91	3.61 (2.40)	1.15
(14,10)	4.46 (3.09)	**4.94**	3.37	3.75 (2.50)	1.20
(15,10)	3.67 (2.47)	**4.19**	2.54	3.07 (1.99)	0.96
(16,10)	2.82 (2.04)	**3.69**	2.98	2.35 (1.62)	0.80

to the restriction of JE-128 and ISA-L on the parameter w, we also provide the results for $w = 8$ for Jo-CRS and Jo-CRS-128, which are given in the parentheses.

- In Table 3.23, the proposed approach is compared with well-known codes with three parities.
- In Table 3.24, the proposed approach is compared with well-known codes with two parities.

It is important to first note from Table 3.15 - 3.18 that the approach of vectorizing finite field operations (JE-128 and ISA-L) is in general inferior to the strategies based on vectorizing XOR operations (Jo-CRS and CRS). This is true even if we restrict the XOR-based vectorization in the proposed approach to 128-bit (Jo-CRS-128 vs. ISA-L and JE-128), instead of 256-bit or 512-bit; moreover, it is also true even if we restrict the finite field to be $w = 8$ in the proposed approach

Table 3.16: Encoding throughput (GB/s) for methods that allow general (n, k) parameters (amd), reprinted with permission from [3]

(n, k)	Jo-CRS	CRS	ISA-L	Jo-CRS -128	JE-128
(7,5)	**7.93** (7.39)	7.51	4.36	7.68 (7.12)	2.80
(8,6)	**7.92** (7.42)	7.86	4.10	7.62 (7.13)	2.77
(9,7)	**8.19** (7.43)	7.65	4.68	8.04 (7.30)	2.84
(10,8)	**8.17** (7.13)	7.85	4.48	7.87 (6.90)	2.96
(12,10)	**8.15** (7.47)	7.74	5.07	7.91 (6.97)	3.01
(8,5)	**5.70** (4.86)	5.26	2.89	5.53 (4.67)	1.85
(9,6)	**5.92** (4.84)	5.37	2.90	5.76 (4.63)	1.88
(10,7)	**6.03** (4.77)	5.29	3.09	5.83 (4.57)	1.90
(11,8)	**6.04** (4.82)	5.50	3.22	5.84 (4.59)	1.95
(13,10)	**6.05** (4.82)	5.30	3.51	5.87 (4.58)	2.02
(10,6)	**4.60** (3.51)	4.06	2.39	4.43 (3.38)	1.40
(11,7)	**4.70** (3.57)	4.10	2.74	4.52 (3.41)	1.45
(12,8)	**4.69** (3.63)	4.22	2.74	4.50 (3.46)	1.45
(14,10)	**4.67** (3.67)	4.16	3.11	4.47 (3.49)	1.52
(15,10)	**3.83** (2.93)	3.44	2.29	3.69 (2.80)	1.19
(16,10)	**3.26** (2.41)	2.95	1.92	3.13 (2.28)	1.00

(the data in parentheses for Jo-CRS vs. ISA-L, as well as the data in parentheses for Jo-CRS-128 vs. JE-128).

From Table 3.15 - 3.18, it is also seen that Jo-CRS is faster than all other references on three out of the four platforms. On the (older version) desktop i7 platform, CRS is fast than Jo-CRS. The main reason for this exception appears to be the interaction between cache and the choice of the *packet_size* parameters. As mentioned earlier, Jo-CRS involves more intermediate computation results, which need to be stored in the cache (memory), and thus the *packet_size* will be chosen at a smaller value than CRS. Small *packet_size* tends to penalize throughput, however, the penalty is less significant on newer platforms with a larger cache size. Indeed, on all the other three platforms, the cache is more abundant, and thus the proposed approach performs better than CRS.

Table 3.17: Encoding throughput (GB/s) for methods that allow general (n, k) parameters (xeon), reprinted with permission from [3]

(n, k)	Jo-CRS	CRS	ISA-L	Jo-CRS -128	JE-128
(7,5)	4.48 (**4.82**)	3.62	2.71	3.71 (3.48)	1.31
(8,6)	4.67 (**4.94**)	3.85	2.99	3.79 (3.53)	1.75
(9,7)	4.92 (**5.15**)	3.88	3.22	3.89 (3.73)	1.52
(10,8)	74.91 (**4.92**)	3.95	3.42	3.95 (3.63)	1.57
(12,10)	**5.17** (5.08)	4.12	4.22	4.05 (3.80)	1.69
(8,5)	3.44 (**3.48**)	2.69	1.92	2.72 (2.42)	0.89
(9,6)	**3.65** (3.56)	2.77	2.14	2.80 (2.49)	0.95
(10,7)	**3.81** (3.65)	2.82	2.31	2.92 (2.49)	1.00
(11,8)	**3.99** (3.70)	2.96	2.49	3.00 (2.50)	1.04
(13,10)	5**4.21** (3.67)	3.01	2.99	3.07 (2.50)	1.11
(10,6)	**3.00** (2.74)	2.17	1.66	2.24 (1.83)	0.71
(11,7)	**3.17** (2.78)	2.25	1.82	2.34 (1.87)	0.75
(12,8)	**3.38** (2.83)	2.36	1.96	2.38 (1.88)	0.78
(14,10)	**3.52** (2.96)	2.46	2.38	2.47 (1.92)	0.83
(15,10)	**3.00** (2.43)	2.08	1.69	2.06 (1.55)	0.66
(16,10)	**2.59** (2.05)	1.81	1.50	1.77 (1.27)	0.55

In Table 3.23, the more generic approach of Jo-CRS and CRS using bitmatrix are able to compete with specially designed three-parity codes in some cases, but in other cases, specially designed codes perform better, depending on the specific platform. This confirms the effectiveness of the bitmatrix based approach with vectorized XOR operations, despite its generality. For the case of two parities, Table 3.24 shows that specially design codes do have an advantage over the more generic approach of Jo-CSR and CSR in most cases.

In Table 3.25, we summarize 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 most existing techniques, some by a large margin. The result in this table is included here to provide a summary on the performance by

Table 3.18: Encoding throughput (GB/s) for methods that allow general (n, k) parameters (arm), reprinted with permission from [3]

(n,k)	Jo-CRS	CRS	ISA-L
(7,5)	**2.53** (2.23)	2.49	1.31
(8,6)	**2.63** (2.23)	2.51	1.28
(9,7)	**2.58** (2.21)	2.43	1.44
(10,8)	**2.59** (2.14)	2.54	1.43
(12,10)	**2.53** (2.14)	2.39	1.47
(8,5)	**1.77** (1.43)	1.68	0.86
(9,6)	**1.82** (1.38)	1.65	0.91
(10,7)	**1.79** (1.33)	1.62	0.93
(11,8)	**1.85** (1.28)	1.73	0.94
(13,10)	**1.76** (1.20)	1.64	0.96
(10,6)	**1.37** (0.96)	1.22	0.68
(11,7)	**1.34** (0.92)	1.23	0.69
(12,8)	**1.33** (0.91)	1.30	0.69
(14,10)	**1.30** (0.86)	1.26	0.72
(15,10)	**1.06** (0.69)	1.03	0.57
(16,10)	**0.90** (0.56)	0.87	0.47

various techniques, however for individual (n, k, w) parameter, the performance may vary considerably as indicated by Table 3.15 - 3.18, 3.23, and 3.24.

It is worth emphasizing that in the performance evaluation, we have chosen the *packet_size* according to the rule given in Section 3.3.5, which in fact approximately optimizes the throughput for each strategy. If we were to fix the same *packet_size* across all the strategies, for example at 4K bytes or 16K bytes (keeping the cache usage constraint (3.1) satisfied for all cases), then the advantage of the proposed strategy Jo-CRS would be even more pronounced. In fact, this was the evaluation approach used in our earlier work [4]. Given the importance of the *packet_size* parameter, we have decided to take the updated evaluation approach in this work.

Table 3.19: Decoding throughput (GB/s) for methods that allow general (n, k) parameters (i7), reprinted with permission from [3]

(n, k)	Jo-CRS	CRS	ISA-L	Jo-CRS -128	JE-128
(7,5)	**7.73** (6.55)	7.16	4.55	6.81 (5.30)	2.20
(8,6)	**8.00** (5.83)	7.94	4.48	6.88 (4.49)	2.25
(9,7)	7.58 (5.54)	**7.89**	4.74	6.44 (4.68)	2.29
(10,8)	**7.75** (5.61)	7.69	4.99	6.35 (4.36)	2.32
(12,10)	7.67 (5.79)	**7.89**	6.21	5.55 (4.36)	2.41
(8,5)	**6.19** (3.13)	5.40	3.26	4.84 (2.24)	1.46
(9,6)	5.38 (3.30)	**5.63**	3.30	3.78 (2.29)	1.14
(10,7)	5.45 (3.21)	**5.80**	3.50	3.95 (2.22)	1.53
(11,8)	**5.57** (3.26)	5.48	3.77	3.84 (2.20)	1.17
(13,10)	5.52 (3.11)	**5.90**	4.23	4.02 (2.14)	1.62
(10,6)	4.44 (2.59)	**4.64**	2.55	3.24 (1.59)	1.14
(11,7)	4.64 (2.60)	**4.73**	2.89	3.24 (1.61)	1.16
(12,8)	4.59 (2.57)	**4.50**	3.01	3.24 (1.56)	1.17
(14,10)	4.53 (2.57)	**4.71**	3.61	3.19 (1.70)	1.22
(15,10)	3.83 (2.10)	**4.02**	2.57	2.62 (1.39)	0.99
(16,10)	2.90 (1.78)	**3.57**	2.34	1.98 (1.14)	0.82

Table 3.20: Decoding throughput (GB/s) for methods that allow general (n, k) parameters (amd), reprinted with permission from [3]

(n, k)	Jo-CRS	CRS	ISA-L	Jo-CRS -128	JE-128
(7,5)	7.55 (6.56)	**7.67**	4.36	7.33 (6.26)	2.69
(8,6)	7.45 (6.41)	**7.86**	4.15	7.56 (6.26)	2.75
(9,7)	7.18 (6.03)	**7.71**	4.70	7.58 (5.51)	2.80
(10,8)	7.64 (6.52)	**7.81**	4.43	7.52 (6.01)	2.87
(12,10)	**7.72** (6.57)	7.71	5.07	7.49 (6.30)	2.92
(8,5)	4.63 (3.03)	**5.54**	2.88	5.34 (2.88)	1.77
(9,6)	4.72 (3.02)	**5.09**	2.96	4.55 (2.90)	1.84
(10,7)	**4.94** (3.00)	4.93	3.09	4.78 (2.80)	1.88
(11,8)	5.03 (3.04)	**5.07**	3.26	4.85 (2.86)	1.92
(13,10)	5.06 (2.96)	**5.24**	3.50	4.85 (2.86)	1.96
(10,6)	3.91 (2.36)	**4.04**	2.42	3.71 (2.27)	1.40
(11,7)	3.88 (2.47)	**3.98**	2.75	3.79 (2.34)	1.39
(12,8)	**4.06** (2.43)	3.91	2.78	3.85 (2.33)	1.41
(14,10)	3.91 (2.45)	**3.91**	3.18	3.75 (2.32)	1.46
(15,10)	3.22 (1.98)	**3.28**	2.40	3.15 (1.88)	1.18
(16,10)	2.79 (1.67)	**2.81**	1.97	2.65 (1.61)	0.99

Table 3.21: Decoding throughput (GB/s) for methods that allow general (n, k) parameters (xeon), reprinted with permission from [3]

(n, k)	Jo-CRS	CRS	ISA-L	Jo-CRS -128	JE-128
(7,5)	**3.90** (3.58)	3.67	2.71	3.29 (2.59)	1.44
(8,6)	**4.07** (3.82)	3.86	2.99	3.39 (2.65)	1.56
(9,7)	**4.08** (3.93)	3.87	3.21	3.28 (2.78)	1.63
(10,8)	**4.19** (4.11)	4.00	3.39	3.29 (2.78)	1.70
(12,10)	**4.48** (4.28)	4.21	4.20	3.26 (2.85)	1.78
(8,5)	**2.89** (2.38)	2.69	1.91	2.37 (1.45)	0.99
(9,6)	**2.71** (2.64)	2.68	2.13	2.11 (1.50)	1.05
(10,7)	**2.96** (2.70)	2.79	2.32	2.22 (1.48)	1.10
(11,8)	**3.18** (2.62)	2.84	2.47	2.23 (1.43)	1.13
(13,10)	**3.47** (2.91)	3.00	2.96	2.27 (1.45)	1.20
(10,6)	**2.30** (2.17)	2.13	1.67	1.74 (1.18)	0.76
(11,7)	**2.55** (2.33)	2.21	1.82	1.75 (1.18)	0.83
(12,8)	**2.73** (2.36)	2.28	1.96	1.80 (1.18)	0.85
(14,10)	**2.95** (2.41)	2.43	2.38	1.85 (1.18)	0.90
(15,10)	**2.74** (2.08)	2.03	1.68	1.63 (0.97)	0.72
(16,10)	**2.39** (1.78)	1.76	1.51	1.39 (0.83)	0.60

Table 3.22: Decoding throughput (GB/s) for methods that allow general (n, k) parameters (arm), reprinted with permission from [3]

(n, k)	Jo-CRS	CRS	ISA-L
(7,5)	**2.62** (1.91)	2.55	1.30
(8,6)	**2.67** (1.97)	2.54	1.36
(9,7)	**2.50** (1.77)	2.48	1.40
(10,8)	**2.50** (1.78)	2.34	1.43
(12,10)	2.43 (1.68)	**2.50**	1.49
(8,5)	1.65 (0.87)	**1.75**	0.86
(9,6)	1.48 (0.85)	**1.58**	0.90
(10,7)	1.52 (0.81)	**1.57**	0.92
(11,8)	**1.53** (0.78)	1.43	0.94
(13,10)	1.48 (0.74)	**1.57**	0.97
(10,6)	1.15 (0.62)	**1.15**	0.66
(11,7)	1.15 (0.61)	**1.18**	0.68
(12,8)	**1.18** (0.59)	1.07	0.69
(14,10)	1.11 (0.56)	**1.19**	0.72
(15,10)	0.91 (0.45)	**1.00**	0.57
(16,10)	0.79 (0.37)	**0.84**	0.47

Table 3.23: Encoding throughputs (GB/s): Three parities, reprinted with permission from [3]

	i7				amd				xeon				arm			
(n,k)	Jo-CRS	CRS	QFS	STAR	Jo-CRS	CRS	QFS	STAR	Jo-CRS	CRS	QFS	STAR	Jo-CRS	CRS	QFS	STAR
(8,5)	**5.37**	5.23	4.02	5.24	**5.70**	5.26	4.18	5.58	**3.44**	2.69	2.84	3.39	1.77	1.68	**1.88**	1.29
(9,6)	5.52	**6.06**	4.14	5.56	5.92	5.37	4.26	**5.95**	**3.65**	2.77	2.99	3.54	1.82	1.65	**2.04**	1.29
(10,7)	5.31	**5.99**	4.25	5.85	6.03	5.29	4.30	**6.19**	**3.81**	2.82	3.11	3.68	1.79	1.62	**2.07**	1.28
(11,8)	5.59	**6.08**	4.40	5.90	6.04	5.50	4.32	**6.38**	3.99	2.96	3.21	**4.19**	1.85	1.73	**1.97**	1.28
(12,9)	5.19	**6.17**	4.60	6.12	6.10	5.44	4.34	**6.53**	4.11	3.01	3.29	**4.34**	1.79	1.69	**2.00**	1.27
(13,10)	5.49	5.96	4.65	**6.15**	6.05	5.30	4.37	**6.49**	4.21	3.01	3.36	**4.39**	1.76	1.64	**1.94**	1.27

Table 3.24: Encoding throughputs (GB/s): Two parities, reprinted with permission from [3]

	i7					amd					xeon					arm				
(n,k)	Jo-CRS	CRS	EO	RDP	Raid6	Jo-CRS	CRS	EO	RDP	Raid6	Jo-CRS	CRS	EO	RDP	Raid6	Jo-CRS	CRS	EO	RDP	Raid6
(7,5)	6.94	7.02	**7.36**	7.26	5.87	7.93	7.51	7.03	**8.10**	6.91	4.48	3.62	3.62	**5.04**	4.06	2.53	2.49	2.57	**3.09**	2.29
(8,6)	6.98	**7.94**	7.83	7.68	6.23	7.92	7.86	7.41	**8.38**	7.07	4.67	3.85	3.83	**5.24**	4.30	2.63	2.51	2.77	**3.13**	2.29
(9,7)	7.20	7.92	**8.08**	7.75	6.38	8.19	7.69	7.87	**8.67**	7.09	4.92	3.88	4.03	**5.55**	4.47	2.58	2.43	2.84	**2.87**	2.31
(10,8)	7.13	7.98	**8.13**	7.99	6.53	8.17	7.85	8.08	**8.96**	7.27	4.91	3.95	4.40	**5.70**	4.63	2.59	2.54	2.68	**2.86**	2.32
(11,9)	7.22	7.89	**8.34**	8.17	6.72	8.25	7.53	8.28	**9.12**	7.32	5.03	3.98	4.57	**5.73**	4.73	2.58	2.39	2.65	**2.80**	2.33
(12,10)	7.12	7.92	**8.40**	8.26	6.94	8.15	7.74	8.61	**9.34**	7.42	5.17	4.12	4.73	**5.93**	4.82	2.53	2.48	2.67	**2.75**	2.33

3.4.3 Decoding Performance

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 useful to understand the impact of optimizing the encoding bitmatrix and procedure on the decoding performance. 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 3.19 - 3.22, Jo-CRS and CRS in most cases provide similar throughputs, and both are better than vectorized GF-based RS code, i.e., ISA-L and JE-128. This suggests that the

Table 3.25: Encoding throughput improvements over references, reprinted with permission from [3]

Reference codes or methods	Improvement by proposed code			
	i7	amd	xeon	arm
General (n, k) Codes				
GF-based RS code w/o vectorization	1131.71%	950.51%	1106.54%	439.86%
XOR-based CRS code w/o vectorization	87.93%	85.20%	112.85%	56.60%
Vectorized GF-based RS code [28]	257.14%	188.37%	282.48%	93.56%
Vectorized XOR-based CRS code	-9.44%	4.85%	34.76%	8.50%
Intel® ISA-L[33]	49.26%	73.82%	58.52%	-
Three Parities Codes				
STAR [25]	-5.29%	-5.79%	-1.08%	-8.05%
Quantcast-QFS [12]	26.48%	35.51%	23.18%	43.34%
Two Parities Codes				
Raid-6 w/o vectorization	384.36%	337.22%	481.84%	846.60%
Vectorized Raid-6	11.96%	9.86%	9.71%	13.62%
RDP [24]	-8.00%	-9.83%	-10.75%	-9.77%
EVENODD [23]	-10.15%	0.64%	18.19%	-2.50%

proposed approach, despite using a bitmatrix optimized for encoding, does not suffer significantly in decoding throughput. In contrast to encoding, CRS is more competitive in decoding, which is mainly due to the advantage in the larger *packet_size* values. For codes with three parities, it can be seen from Table 3.26 that Jo-CRS performs well, and again sometimes can compete with codes in the literature specifically designed for this case. For codes with two parities, as shown in Table 3.27, the decoding throughput of proposed approach is usually lower than EVENODD and RDP codes.

3.5 Conclusion

We performed a comprehensive study of the erasure coding acceleration techniques in the literature. Several acceleration techniques are evaluated, combined and jointly optimized. The study led us to a simple procedure: produce a computation schedule based on an optimized bitmatrix, together with the BN and WM (or UM) technique, then use vectorized XOR operation in the computation schedule. The proposed approach outperforms most existing approaches, particularly

Table 3.26: Decoding throughputs (GB/s): Three parities, reprinted with permission from [3]

	i7				amd				xeon				arm			
(n, k)	Jo-CRS	CRS	QFS	STAR	Jo-CRS	CRS	QFS	STAR	Jo-CRS	CRS	QFS	STAR	Jo-CRS	CRS	QFS	STAR
(8,5)	**6.19**	5.40	4.11	4.52	4.63	**5.54**	4.89	5.08	**2.89**	2.69	2.95	2.34	1.65	**1.75**	1.54	1.41
(9,6)	5.38	**5.63**	4.22	4.27	4.72	**5.09**	5.04	4.30	2.71	2.68	2.93	**3.11**	1.48	**1.58**	1.54	1.40
(10,7)	5.45	**5.80**	4.19	4.03	4.94	4.93	5.13	**5.60**	2.96	2.79	2.86	**3.34**	1.52	**1.57**	1.54	1.40
(11,8)	**5.57**	5.48	4.20	3.89	5.03	**5.07**	5.18	4.77	**3.18**	2.84	2.84	3.13	**1.53**	1.43	1.41	1.39
(12,9)	5.55	**5.66**	4.38	4.96	4.96	4.61	4.34	**6.53**	**3.41**	2.92	2.83	2.89	1.53	**1.60**	1.37	1.39
(13,10)	5.52	**5.90**	4.31	4.51	5.06	5.24	**5.31**	4.92	**3.47**	3.00	2.81	3.09	1.48	**1.57**	1.46	1.38

Table 3.27: Decoding throughputs (GB/s): Two parities, reprinted with permission from [3]

	i7					amd					xeon					arm				
(n, k)	Jo-CRS	CRS	EO	RDP	Raid6	Jo-CRS	CRS	EO	RDP	Raid6	Jo-CRS	CRS	EO	RDP	Raid6	Jo-CRS	CRS	EO	RDP	Raid6
(7,5)	7.73	7.16	**8.15**	7.79	6.09	7.55	7.67	8.15	**8.65**	7.23	3.90	3.66	**5.61**	5.38	4.07	2.62	2.55	3.16	**3.72**	2.27
(8,6)	**8.00**	7.94	7.89	7.89	6.17	7.45	7.86	**9.56**	9.52	7.44	4.07	3.86	**5.81**	5.78	4.30	2.67	2.54	3.04	**3.62**	2.24
(9,7)	7.58	7.89	**8.65**	7.25	6.25	7.18	7.71	**10.43**	10.40	7.59	4.08	3.87	5.31	**5.97**	4.30	2.50	2.45	3.19	**3.39**	2.25
(10,8)	7.75	7.69	**8.07**	7.11	6.31	7.64	7.81	**10.11**	9.36	7.73	4.19	4.00	**6.45**	6.12	4.66	2.50	2.34	**3.07**	2.88	2.28
(11,9)	7.67	**8.04**	7.30	7.64	6.41	7.74	7.54	9.29	**9.31**	7.86	4.37	4.08	6.38	**6.40**	4.64	2.54	2.48	2.92	**3.07**	2.29
(12,10)	7.67	7.89	**8.22**	6.28	6.47	7.72	7.71	8.80	**10.60**	8.03	4.48	4.21	**6.49**	5.94	4.62	2.43	2.50	**2.92**	2.90	2.28

when the number of parity is greater than two. One particularly important outcome 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.

4. DELAYED PARITY GENERATION [1]

4.1 Relevant Results on Regenerating Codes

MDS codes, such as Reed-Solomon codes [36], can be specified by two parameters: the number of data nodes k and the number of parity nodes m; such codes are usually referred to as $(k+m,k)$ erasure codes. The most important property of MDS codes is that they can withstand a loss of less than or equal to m nodes, which makes them particularly suitable for data storage purpose.

Two special classes of MDS erasure codes, referred to as minimum storage regenerating (MSR) codes and MDS codes with minimum rebuilding access, have attracted significant recent attention [38, 39, 40, 41, 44, 42, 45]. In such codes, any node failures can be repaired efficiently by downloading, or accessing, the minimum amount of data from the remaining nodes. Such codes are designed to allow more efficient node repairs in distributed data storage systems, since less data traffic and less data access imply more efficient repair of failed nodes.

The minimum amount of total data access (and data download), normalized by the size of the raw data, when repairing r failed nodes ($r \leq m'$) jointly in a $(k+m',k)$ regenerating code system, and all the $k+m'-r$ remaining nodes are allowed to participate in the repair, is given as (see [38, 39])

$$\bar{\gamma}^* = \frac{r(m'+k-r)}{km'}. \tag{4.1}$$

It was shown in [56] that for regenerating codes with optimal rebuilding access, the subpacketization factor (assuming $r=1$) is bounded below by

$$\alpha \geq \min(m'^{\lceil \frac{k+m'-1}{m'} \rceil}, m'^{k-1}). \tag{4.2}$$

For high rate code, this subpacketization factor can be rather significant.

[1] ©2018 IEEE. Partly Reprinted, with permission, from S. Mousavi, T. Zhou and C. Tian, *Delayed parity generation in MDS storage codes*, 2018 IEEE International Symposium on Information Theory (ISIT), June 2018

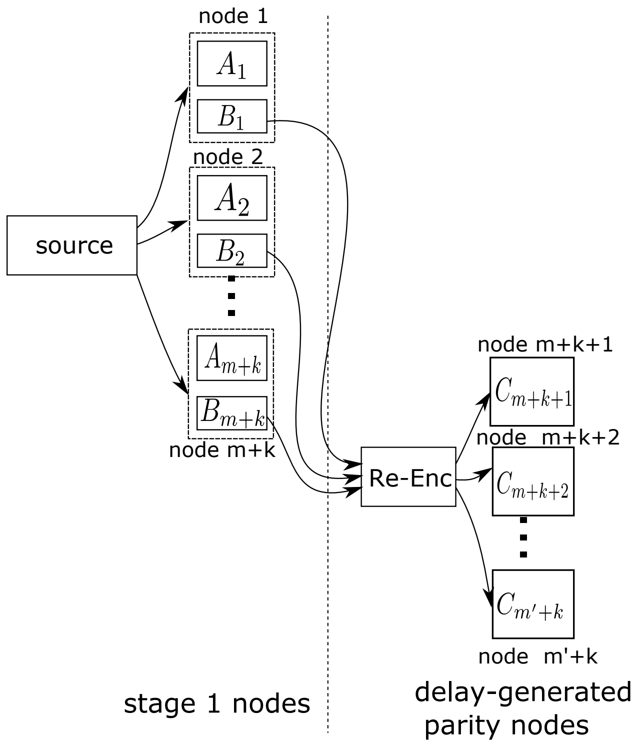

Figure 4.1: System model for delayed parity generation. The node labeled "Re-Enc" corresponds to the processing center in charge of the delayed parity generation. B_i's stand for the parts of the shares being accessed in the second stage, while A_i's are the parts that not being accessed. C_i's are the delay-generated parity nodes, reprinted with permission from [1]

4.2 Delayed Parity Generation: The System Model

A system using delayed parity generation, as its name suggests, operates in two stages:

1. In the first stage, the B units of raw data is first divided into k shares of equal size $\alpha = B/k$, which are then used to produce m parity shares of size $\alpha = B/k$ each. The coding requirement at this stage is that any k shares in the total $(k+m)$ shares can be used to recover the original data.

2. In the second stage, from each of the existing $(k + m)$ shares, β units of data is accessed, which are then collected to a processing center. From the downloaded data, the processing center computes the $(m' - m)$ new parity shares of size $\alpha = B/k$ each. The coding requirement here is that any k out of the total $k + m'$ shares ($k + m$ generated in the first stage and

$m' - m$ generated in the second stage) can be used to recover the original data.

The two-stage encoding process is illustrated in Fig. 4.1, and we refer to it as (k, m, m') delayed parity generation.

The performance of a valid coding strategy is measured by the total amount of data access $\gamma \triangleq (k+m)\beta$ during DPG. Since the total amount of initial data is B, and there is a linear relation between B and γ, we can equivalently consider the normalized data access $\bar{\gamma} = \frac{\gamma}{B}$. Allowing different amounts of data accesses from nodes in the second stage cannot improve the optimal value of $\bar{\gamma}$, and thus the uniform-access-amount model above is without loss of optimality; this can be shown straightforwardly through symmetrizing any code with non-uniform data access, by way of space-sharing different placement patterns of the data shares and parity shares.

In this problem setting, we focus on the amount of *data access* γ, instead of the amount of *data download* from the existing nodes, the former of which is clearly more stringent. Our result given later shows that the two measures in fact reduce to the same, *i.e.*, an optimal code allows simply transmitting from each existing node what is being accessed, without any need for further computation before transmission. This is beneficial in terms of system implementation, since complex computation is eliminated at the helper nodes in the second stage, *i.e.,* it has the help-by-transfer property.

4.3 The Fundamental Limit of Delayed Parity Generation

The theorem given below provides the first main theoretical result on delayed parity generation.

Theorem 1. (k, m, m') *delayed parity generation can be accomplished if and only if*

$$\bar{\gamma} \geq \min\left\{1, \frac{(m'-m)(k+m)}{km'}\right\}. \tag{4.3}$$

As mentioned earlier, this result is established through a connection to multicast network coding. This approach is along the same vein taken in [38]. However, because of the relaxed problem setting, the bounds derived in [38] do not apply directly, and new bounds are needed in our setting.

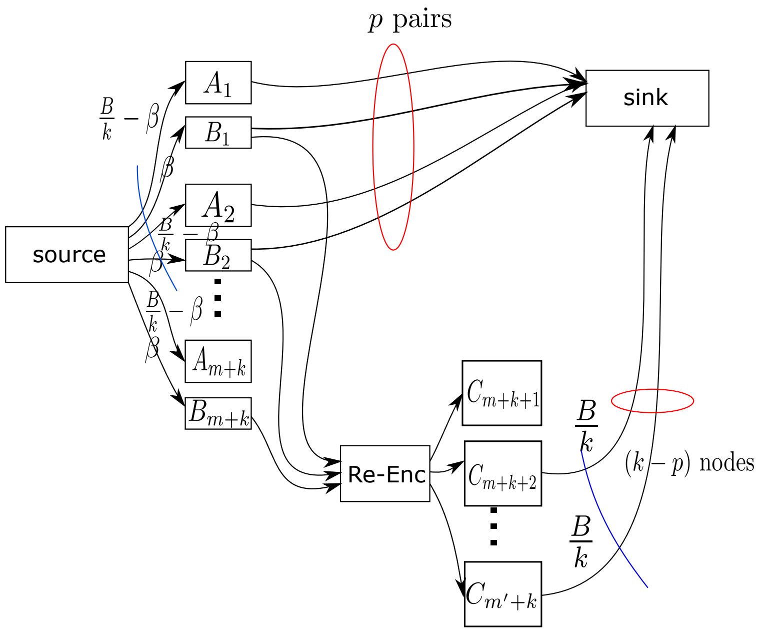

Figure 4.2: The graphical model used in the proof of Theorem 1, reprinted with permission from [1]

Proof. Let us consider a graphical representation of the problem as illustrated in Fig. 4.2. There are $(m + k)$ type A nodes, which represent the parts in the $(m + k)$ coded shares produced in the first stage that are not accessed during the second stage (size $B/k - \beta$ each); the $(m + k)$ type B nodes represent the parts in the original $(m + k)$ coded shares that are accessed (β each) during the second stage; the $(m' - m)$ type C nodes are the new parity shares (B/k each) generated in the second stage. All non-labeled links have infinite capacity.

Because of the coding requirements in the two stages as stated in the previous sections, we can view this system as a multicast network. There are multiple possible sinks: each possible sink is connected to a specific combination of $p \leq k$ ($p \geq 0$) pairs of type A and type B nodes, and $(k-p)$ type C nodes; *i.e.,* any k storage shares can be used to recover the data content. Clearly there are a total of

$$\sum_{p=\max(0,k-(m'-m))}^{k} \binom{k+m}{p}\binom{m'-m}{k-p} \tag{4.4}$$

different sinks in this multicast network. It follows from the well known network coding multicast result [34, 35] that there exist linear codes to fulfill all the requirements, if and only if the min-cut between the source and any one of possible sinks is greater than or equal to B.

When $k \geq m' - m$, the parameter p must satisfy $k - m' + m \leq p \leq k$, and the min-cut condition is equivalent to

$$
\begin{aligned}
B &\leq \min_p \left\{ p\frac{B}{k} + \min\left[(k-p)\frac{B}{k}, (k+m-p)\beta \right] \right\} && (4.5)\\
&= \min\left\{ B, \min_p \left[p\frac{B}{k} + (k+m-p)\beta \right] \right\} \\
&= \min\left\{ B, (k+m)\beta + \min_p p\left[\frac{B}{k} - \beta \right] \right\} \\
&= \min\left\{ B, (k+m)\beta + (k-m'+m)\left[\frac{B}{k} - \beta \right] \right\}, && (4.6)
\end{aligned}
$$

where the outer minimization over p in (4.5) is to take into account all the possible sinks linked through different combinations of (A, B) pairs and C nodes. The inner minimization in (4.5) is to cut the paths through type B nodes, which can include the edges either before or after the re-enc node: for the former, all the edges connecting to the type B nodes need to be included in the cut, and for the latter, the $(k - p)$ nodes connecting to the sink need to be included in the cut. The inequality above can then be written as

$$
(k+m)\beta + (k-m'+m)\left[\frac{B}{k} - \beta \right] \geq B, \tag{4.7}
$$

which further simplifies to

$$
\gamma = (k+m)\beta \geq \frac{(m'-m)(k+m)B}{km'}. \tag{4.8}
$$

On the other hand, when $k < m' - m$, the parameter p must satisfy $0 \leq p \leq k$, and the min-cut requirement can be similarly simplified to

$$\gamma = (k+m)\beta \geq B. \tag{4.9}$$

Combining the two cases in (4.8) and (4.9) together and normalizing both sides give the desired bound. □

As mentioned earlier, regenerating codes can be used for delayed parity generation although they have functions more than what are required here. It is worth noting that in this case the bound on $\bar{\gamma}$ in Theorem 1 indeed coincides with the corresponding minimum repair bandwidth derived for regenerating codes in [38] and [39]. This can be seen by setting $r = m' - m$ in (4.1). This equality is in fact somewhat surprising, given the much relaxed problem setting. Nevertheless, the fact that the normalized data access in delayed parity generation has the same optimal value as the normalized repair bandwidth in regenerating codes does not necessarily mean using regenerating codes for delayed parity generation is a good choice. This will become evident after we present an explicit code construction in the sequel.

4.4 An Explicit and Low-Complexity Code Construction through Code Transformation

Theorem 1 provides the fundamental limit on the amounts of data access for delayed parity generation, but it does not provide explicit code constructions. It is clear that code constructions for the case when

$$m' \geq k+m \tag{4.10}$$

is straightforward, which corresponds to the degenerate case of $\bar{\gamma} = 1$, *i.e.*, the full set of data is read. For this case, we can start with any $(k+m', k)$ MDS code, but only generate the first m shares among the m' parity shares in the first stage. In the second stage, the remaining $(m'-m)$ parity shares can be generated by reading all the existing data shares[2]. This strategy does not yield

[2]This is equivalent to a punctured code [36].

uniform read loads among the existing storage devices, however we can simply stack multiple such codes to balance the load for the required uniformity, and then use linear transformations to make the overall code systematic. Intuitively speaking, this case corresponds to the situation where too many parities are left to be generated in the second stage, and thus all data must be downloaded when doing so. Note that in this case $m' \geq k$, thus the coding rate is less than half, implying a low-rate erasure code.

For the case when (4.10) does not hold, which is the case of most interest in practice where high-rate erasure codes are more desirable, designing optimal codes is less straightforward. The new parity shares must be generated by reading only partial shares of the existing nodes in the second stage, and these partial shares do not represent the complete stored data and thus the simple strategy for the degenerate case (4.10) does not apply. In the sequel, we present a novel and low-complexity code construction for delayed parity generation for this case. For notation simplicity, we restrict our attention to scalar MDS codes in the discussion given below, however, the generalization to vector codes is rather straightforward.

4.4.1 The Coding Procedure

Consider an arbitrary scalar systematic MDS code $\mathcal{C}$ with parameter $(k+m', k)$. A total of $B = km'$ information symbols in a certain finite field $\mathbb{F}_q$ are used to first generate $m'm$ parity symbols in the first stage, and then $m'(m'-m)$ parity symbols in the second stage. The subpacketization factor in our code is m'.

Conceptually, we first partition the B information symbols into groups of k each, then encode each group using the MDS code $\mathcal{C}$; the i-th group after this encoding is thus the vector

$$[d_1^{(i)}, ..., d_k^{(i)}, p_1^{(i)}, ..., p_{m'}^{(i)}]; \tag{4.11}$$

where the first part is the information symbols $[d_1^{(i)}, ..., d_k^{(i)}]$, and the remaining part is the parity symbols. All the m' groups together can be collected and represented as in Table 4.1, with each column corresponding to a vector in (4.11).

Table 4.1: Coding symbols arranged in a table, reprinted with permission from [1]

systematic-1	$d_1^{(1)}$	$d_1^{(2)}$	$\cdots$	$d_1^{(m')}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
systematic-k	$d_k^{(1)}$	$d_k^{(2)}$	$\cdots$	$d_k^{(m')}$
parity-1	$p_1^{(1)}$	$p_1^{(2)}$	$\cdots$	$p_1^{(m')}$
parity-2	$p_2^{(1)}$	$p_2^{(2)}$	$\cdots$	$p_2^{(m')}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
parity-m'	$p_{m'}^{(1)}$	$p_{m'}^{(2)}$	$\cdots$	$p_{m'}^{(m')}$

Table 4.2: Systematic and parity symbols in the two stages, reprinted with permission from [1]

	systematic-1	$d_1^{(1)}$	$d_1^{(2)}$	$\cdots$	$d_1^{(m)}$	$d_1^{(m+1)}$	$d_1^{(m+2)}$	$\cdots$	$d_1^{(m')}$
	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
	systematic-k	$d_k^{(1)}$	$d_k^{(2)}$	$\cdots$	$d_k^{(m)}$	$d_k^{(m+1)}$	$d_k^{(m+2)}$	$\cdots$	$d_k^{(m')}$
1st stage	parity-1	$p_1^{(1)}$	$p_1^{(2)}$	$\cdots$	$p_1^{(m)}$	$p_1^{(m+1)} + p_{m+1}^{(1)}$	$p_1^{(m+2)} + p_{m+2}^{(1)}$	$\cdots$	$p_1^{(m')} + p_{m'}^{(1)}$
	parity-2	$p_2^{(1)}$	$p_2^{(2)}$	$\cdots$	$p_2^{(m)}$	$p_2^{(m+1)} + p_{m+1}^{(2)}$	$p_2^{(m+2)} + p_{m+2}^{(2)}$	$\cdots$	$p_2^{(m')} + p_{m'}^{(2)}$
	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
	parity-m	$p_m^{(1)}$	$p_m^{(2)}$	$\cdots$	$p_m^{(m)}$	$p_m^{(m+1)} + p_{m+1}^{(m)}$	$p_m^{(m+2)} + p_{m+2}^{(m)}$	$\cdots$	$p_m^{(m')} + p_{m'}^{(m)}$
2nd stage	parity-$(m+1)$	$p_{m+1}^{(1)}$	$p_{m+1}^{(2)}$	$\cdots$	$p_{m+1}^{(m)}$	$p_{m+1}^{(m+1)}$	$p_{m+1}^{(m+2)}$	$\cdots$	$p_{m+1}^{(m')}$
	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
	parity-m'	$p_{m'}^{(1)}$	$p_{m'}^{(2)}$	$\cdots$	$p_{m'}^{(m)}$	$p_{m'}^{(m+1)}$	$p_{m'}^{(m+2)}$	$\cdots$	$p_{m'}^{(m')}$

The parity symbols of the proposed code to be generated in the two stages are given in Table 4.2. It is seen that in the first stage, the parity symbols in the first m columns are simply the original parity symbols of the MDS code $\mathcal{C}$. On the other hand, each entry of the other $m' - m$ parity symbols of these parity nodes are a linear finite field summation of the original symbol and its diagonal-symmetric entry; this summation can be in any finite field to which $\mathbb{F}_q$ is a subfield, and in particular it can be in the binary field when we are working with computer words. The encoding procedure in the first stage is as follows:

1. Generate the parity symbols $p_i^{(j)}$ for $i = 1, 2, \ldots, m$ and $j = 1, 2, \ldots, m'$;

2. Generate the parity symbols $p_i^{(j)}$ for $i = m+1, m+2, \ldots, m'$ and $j = 1, 2, \ldots, m$;

3. Compute $p_i^{(j)} + p_j^{(i)}$ for $i = 1, 2, \ldots, m$ and $j = m+1, m+2, \ldots, m'$ using the symbols computed in step 1) and step 2).

In contrast to the original MDS code $\mathcal{C}$, the additional computation in this first stage is for the block of parities in step 2) in the left lower corner of Table 4.2, and the computation of the additions in step 3). There is no change in the alphabet size from that of code $\mathcal{C}$, and the increase in computation cost is rather minimal.

The parity symbols for the delay-generated parities in the proposed code are exactly the same as the original MDS code $\mathcal{C}$, which however must be generated by reading data and the parities produced in the first stage. The coding procedure is as follows:

1. Read data $d_i^{(j)}$ for $i = 1, 2, \ldots, k$ and $j = m+1, m+2, \ldots, m'$;

2. Read parities $p_i^{(j)} + p_j^{(i)}$ for $i = 1, 2, \ldots, m$ and $j = m+1, m+2, \ldots, m'$;

3. Compute $p_i^{(j)}$ for $i = 1, 2, \ldots, m'$ and $j = m+1, m+2, \ldots, m'$ using the data read in step 1) and the encoding matrix of the MDS code $\mathcal{C}$;

4. Compute $p_i^{(j)}$ for $i = m+1, m+2, \ldots, m'$ and $j = 1, 2, \ldots, m$, by eliminating $p_t^{(r)}$ for $t = 1, 2, \ldots, m$ and $j = m+1, m+2, \ldots, m'$ that were computed in step 3), from the parities which were read in step 2).

Note that in step 3), the needed parities $p_i^{(j)}$ for $i = m+1, m+2, \ldots, m'$ and $j = m+1, m+2, \ldots, m'$ are produced, and in step 4), $p_i^{(j)}$ for $i = m+1, m+2, \ldots, m'$ and $j = 1, 2, \ldots, m$ are produced. Intuitively, a block Gaussian elimination procedure is performed in step 4), using the block matrix structure as given in Table 4.2. The amount of data access in the second stage is clearly given by $\gamma = (m' - m)(k + m)$ with $\beta = (m' - m)$, and since $B = km'$, the normalized data access is

$$\bar{\gamma} = \frac{(m' - m)(k + m)}{km'}, \tag{4.12}$$

which matches the lower bound in Theorem 1 for the case $m' \leq k + m$ and thus is optimal.

4.4.2 Decoding and Data Recovery

We need to show that the two decoding requirements given in Section 4.2 can be satisfied, which is equivalent to the following proposition.

Proposition 1. *The code given in Table 4.2 is a $(k+m', k)$ MDS code, when code $\mathcal{C}$ is a $(k+m', k)$ MDS code.*

Proof. We show that any k out of the $k + m'$ rows in Table 4.2 can be used to recover the raw information data. Suppose $t \geq 0$ systematic nodes, $t_1 \geq 0$ parity generated in the first stage, and $t_2 \geq 0$ parity generated in the second stage are used in the reconstruction, where $t + t_1 + t_2 = k$; denote the set of the available nodes as $\mathcal{A}$. It is clear that with this k rows of data as specified in Table 4.2, all the data in the first m columns can be recovered from the symbols in the rows of $\mathcal{A}$, because code $\mathcal{C}$ is an MDS code (which has the property that any k out of $k + m'$ symbols can be used to recover the data). Using the recovered $p_i^{(j)}$, $i = m+1, m+2, \ldots, m'$ and $j = 1, 2, \ldots, m$, we can eliminate the terms of the first m columns in the $p_i^{(j)} + p_j^{(i)}$ entries of the rows in $\mathcal{A}$. After these eliminations, we have the remaining $m' - m$ columns of Table 4.1 in the k rows of $\mathcal{A}$, in the native clean form. Since code $\mathcal{C}$ is an MDS code, all the data in the last $m' - m$ columns of Table 4.1 can thus be recovered. This implies that with any k rows of data, we can recover all the data $d_i^{(j)}$, $i = 1, 2, \ldots, k$ and $j = 1, 2, \ldots, m'$. The proof is complete. $\square$

We emphasize that in the proposed code, the subpacketization is simply m', while if one uses optimal regenerating codes with the same amount of data access, the subpacketization factor is at least $m'^{\lceil (k+m'-1)/m' \rceil}$. This difference can be significant, for example, when $(k, m') = (10, 4)$ (a popular choice in [57]), the proposed code has a subpacketization factor of 4, whereas the solution based on regenerating codes will have a subpacketization factor of at least $4^4 = 256$. Moreover, the proposed code is much simpler and thus uses less computation resource than known high-rate MSR codes (*e.g.*, [45]).

4.5 Motivating Applications

In this section, we discuss two applications which motivated our interest in the delayed parity generation approach.

Application 1: Tiered data storage. Data tiering involves moving data between hot storage and cold storage. Data in hot storage is usually stored in a replicated form (e.g., 3-way replicated Hadoop) for better data availability and parallelism, and as data becomes cold with less frequent access, it can be moved to erasure coded cold storage. This data migration needs to be completed as soon as possible since hot storage space is usually highly constrained (e.g., a pool of SLC flash drives on Infiniband network). However cold storage is usually built on lower-grade devices and low bandwidth networking (e.g., consumer-grade HHD on 1GbE network) to be cost-effective. Therefore, even with a high quality bridge between the hot side and the cold side, the migration can be slow. Delayed parity generation can be applied here, as the delayed parities are not transmitted or written which expedites the migration, but the first stage parities in the cold side can still offer sufficient failure protection. After the migration is complete and the system idles, the delayed parities can be generated within the cold side for longer term reliability.

Application 2: Logging in data management systems. As modern data management system becomes larger, the amount of logging data grows accordingly. The collected logging data may undergo certain streaming analysis, then written into distributed data storage for further archival or analysis. The property of the logging data stipulates it be written into storage at the same pace as the streaming throughput, right at the time when the system is at high-load. As such, the computation and communication of the parities for the logging data may negatively impact the system. Delaying generating some parities offers an attractive solution. When the system becomes idle, the additional parities for the log can be generated for better long term reliability.

4.6 Parameters in Delayed Parity Generation

Similar to a traditional erasure code which can be specified by two parameters (k, m), a delayed parity generation code is specified by three parameters (k, m, m'). To encode $k \cdot r$ data symbols, where r is the number of stripes, the coding process proceeds in two stages. In the first stage, $m \cdot r$ parity symbols are produced. Each one of the $k + m'$ device is then given 1 symbol in each stripe, either a data symbol or a parity symbol. At this point, the system can withstand a loss of any m devices. In the second stage, a controller reads a subset of the information from the existing devices to generate the remaining $(m' - m) \cdot r$ parity symbols. The new parity symbols in each stripe are then distributed to $m' - m$ other storage devices, such that the system can withstand a loss of m' devices. In order to reduce the data access and improve the efficiency of parity generation in the second stage, a code construction was proposed in [1] which has the optimal data access, and also a simple parity generation procedure.

4.7 BB-DPG: A Generalized DPG Code

We propose a generalization of the DPG method proposed in [1]. The proposed approach has the benefit of being more flexible for subsequent code design. To distinguish the two approaches, we refer to the new approach as BB-DPG (although the bitmatrix is yet to be introduced), while the one in [1] as DPG. Their difference will be discussed in more details after this new approach is presented.

4.7.1 BB-DPG: The Proposed Code

Let $\mathcal{C}$ and $\bar{\mathcal{C}}$ be two storage codes in certain finite field $GF(2^w)$, both with the parameters (k, m'). A total of $k \cdot m'$ data symbols are partitioned into $r = m'$ stripes, which are denoted as $d^{(1)}, d^{(2)}, \ldots, d^{(m')}$, where $d^{(j)} = (d_1^{(j)}, d_2^{(j)}, \ldots, d_k^{(j)})^T$. The encoding process for the first stage proceeds as follows (see Fig. 4.3):

1. For each of the first m stripes, compute the m' parity symbols using the parity coding matrix of $\mathcal{C}$; these parity vectors are denoted as $p^{(1)}, p^{(2)}, \ldots, p^{(m)}$, where each parity vector $p^{(j)} = (p_1^{(j)}, p_2^{(j)}, \ldots, p_{m'}^{(j)})^T$;

2. For the remaining $m' - m$ stripes, compute the first m parity symbols using parity coding matrix of the second code $\bar{\mathcal{C}}$; these are denoted as $\bar{p}^{(m+1)}, \bar{p}^{(m+2)}, \ldots, \bar{p}^{(m')}$, where $\bar{p}^{(m+j)} = (\bar{p}_1^{(m+j)}, \bar{p}_2^{(m+j)}, \ldots, \bar{p}_m^{(m+j)})^T$;

3. Compute $p_i^{(m+j)} = \bar{p}_i^{(m+j)} \oplus p_{m+j}^{(i)}$ for $i = 1, 2, \ldots, m$ and $j = 1, 2, \ldots, m' - m$;

4. For each of the m' stripes, i.e., $j = 1, 2, \ldots, m'$, write the k data symbols $d^{(j)}$ and the first m parity symbols $(p_1^{(j)}, p_2^{(j)}, \ldots, p_m^{(j)})$ into $k + m$ storage devices.

The parity symbols for the second stage can be generated by reading a subset of information stored in the first stage, more precisely, as follows:

1. Read the data symbols and parity symbols of the last $m' - m$ data stripes, i.e., $d^{(m+j)}$ for $j = 1, 2, \ldots, m' - m$, and $p_i^{(m+j)}$ for $i = 1, 2, \ldots, m$ and $j = 1, 2, \ldots, m' - m$;

2. Compute $\bar{p}_i^{(m+j)}$ for $i = 1, 2, \ldots, m'$ and $j = 1, 2, \ldots, m' - m$ using the data $d^{(m+j)}$ read in step-1 and the encoding function of the code $\bar{\mathcal{C}}$;

3. Compute $p_{m+j}^{(i)}$ for $j = 1, 2, \ldots, m' - m$ and $i = 1, 2, \ldots, m$, by eliminating $\bar{p}_i^{(m+j)}$ from $p_i^{(m+j)}$.

4. For each of the first m stripes, distribute the $m' - m$ parity symbols computed in step-3 into $m' - m$ devices;

5. For each remaining $m' - m$ stripe, write the $m' - m$ parity symbols $\bar{p}_i^{(m+j)}$, $i = m + 1, m + 2, \ldots, m'$ computed in step-2 into $m' - m$ devices (other than the devices already having data or parities of the same stripe);

It is not difficult to see that in the second stage we only need to access $(m' - m)(k + m)$ coded symbols, instead of all km' data symbols. For the example with $(k, m, m') = (3, 2, 3)$, we would access 5 coded symbols, instead of the 9 data symbols, with a saving of almost 50% in terms of data access.

4.7.2 An Example

The correctness of BB-DPG can be proved in a similar manner as DPG, which is omitted here. In the sequel, we use the example $(k, m, m') = (3, 2, 3)$ to illustrate the main idea. Three versions

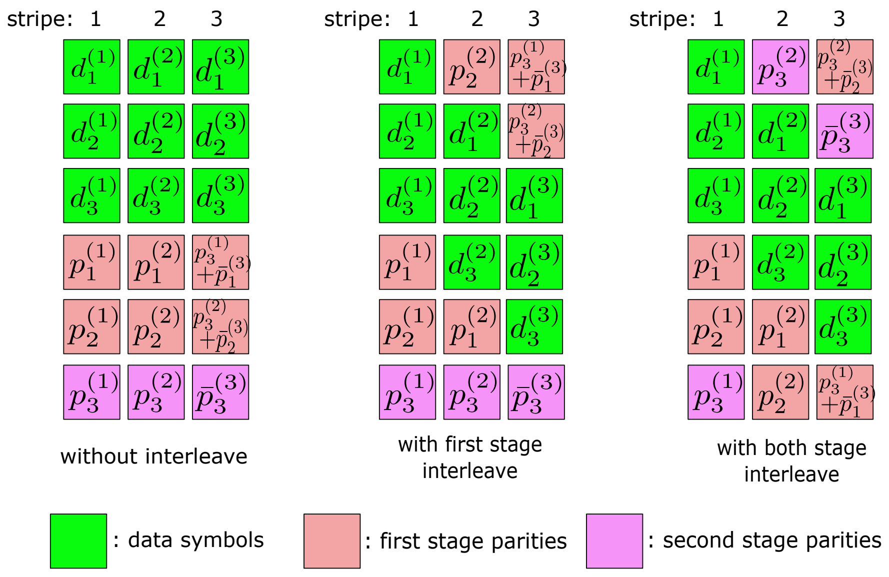
stripe: 1 2 3
$d_1^{(1)}$ $d_1^{(2)}$ $d_1^{(3)}$
$d_2^{(1)}$ $d_2^{(2)}$ $d_2^{(3)}$
$d_3^{(1)}$ $d_3^{(2)}$ $d_3^{(3)}$
$p_1^{(1)}$ $p_1^{(2)}$ $p_3^{(1)}+\bar{p}_1^{(3)}$
$p_2^{(1)}$ $p_2^{(2)}$ $p_3^{(2)}+\bar{p}_2^{(3)}$
$p_3^{(1)}$ $p_3^{(2)}$ $\bar{p}_3^{(3)}$
without interleave
stripe: 1 2 3
$d_1^{(1)}$ $p_2^{(2)}$ $p_3^{(1)}+\bar{p}_1^{(3)}$
$d_2^{(1)}$ $d_1^{(2)}$ $p_3^{(2)}+\bar{p}_2^{(3)}$
$d_3^{(1)}$ $d_2^{(2)}$ $d_1^{(3)}$
$p_1^{(1)}$ $d_3^{(2)}$ $d_2^{(3)}$
$p_2^{(1)}$ $p_1^{(2)}$ $d_3^{(3)}$
$p_3^{(1)}$ $p_3^{(2)}$ $\bar{p}_3^{(3)}$
with first stage interleave
stripe: 1 2 3
$d_1^{(1)}$ $p_3^{(2)}$ $p_3^{(2)}+\bar{p}_2^{(3)}$
$d_2^{(1)}$ $d_1^{(2)}$ $\bar{p}_3^{(3)}$
$d_3^{(1)}$ $d_2^{(2)}$ $d_1^{(3)}$
$p_1^{(1)}$ $d_3^{(2)}$ $d_2^{(3)}$
$p_2^{(1)}$ $p_1^{(2)}$ $d_3^{(3)}$
$p_3^{(1)}$ $p_2^{(2)}$ $p_3^{(1)}+\bar{p}_1^{(3)}$
with both stage interleave
🟩 : data symbols
🟥 : first stage parities
🟪 : second stage parities

Figure 4.3: An example of the proposed code for $(k, m.m') = (3, 2, 3)$. Each row indicates a different storage device.

of the code are shown in Fig. 4.3, one without interleaving, while the other two with different types of interleaving.

Let us consider the version without interleaving first, and there are three requirements:

- The first 5 rows can tolerate a loss of any 2 rows. Let us consider the case when the first two rows are lost. Since the base code $\mathcal{C}$ can tolerate a loss of any two symbols, we can recover $d_1^{(1)}$ and $d_2^{(1)}$ by using $(d_3^{(1)}, p_1^{(1)}, p_2^{(1)})$; similarly, $d_1^{(2)}$ and $d_2^{(2)}$ can be recovered using $(d_3^{(2)}, p_1^{(2)}, p_2^{(2)})$. With $d^{(1)}$ and $d^{(2)}$, we can now compute $p_3^{(1)}$ and $p_3^{(2)}$, and subtract them from $p_3^{(1)} + \bar{p}_1^{(3)}$ and $p_3^{(2)} + \bar{p}_2^{(3)}$, respectively. This gives us $(\bar{p}_1^{(3)}, \bar{p}_2^{(3)})$, and together with $d_3^{(3)}$, we can now recover $d_1^{(3)}$ and $d_2^{(3)}$ since the code $\bar{\mathcal{C}}$ can also tolerate a loss of two symbols. It is straightforward to verify that for other loss patterns, the data can also be recovered.

- The 6 rows together can tolerate a loss of any 3 rows. The idea is similar to the case of the previous case, and we do not elaborate further here.

- The second stage parities (light purple part in 4.3) can be built by reading the first 5 symbols in the last column (green and light red parts in Fig. 4.3). With $(d_1^{(3)}, d_2^{(3)}, d_3^{(3)})$, we can compute $(\bar{p}_1^{(3)}, \bar{p}_2^{(3)}, \bar{p}_3^{(3)})$ using the encoding function of $\bar{\mathcal{C}}$. Subtracting $\bar{p}_1^{(3)}$ and $\bar{p}_2^{(3)}$ from $p_3^{(1)} + \bar{p}_1^{(3)}$ and $p_3^{(2)} + \bar{p}_2^{(3)}$, respectively, we obtain $p_3^{(1)}$ and $p_3^{(2)}$. Thus the second stage parities have all been generated.

Now consider the case with first stage interleaving, which is shown in the middle of Fig. 4.3. To see that the first 5 rows can tolerate a loss of any 2, take the example when the 4-th and 5-th rows are lost. Using the decoding and encoding functions of $\mathcal{C}$, we can recover any symbols in the first 2 columns. Data $d^{(3)}$ is then recovered in a similar manner as in the previous case. It is also not difficult to verify that the 6 rows together can tolerate a loss of any 3, and the second stage parities can be generated in the same manner as before.

Next let us turn the attention to the last case shown on the right of Fig. 4.3, when the data and the first stage parities are in fact distributed to 6 different devices. After the first stage, a loss of any two devices induces a maximum loss of 2 symbols in the first two columns. Thus all the symbols in the first two columns can be obtained using the coding functions of $\mathcal{C}$, and the rest of the procedure follows similarly as before. It is not difficult to verify the second stage parity generation, as well as the data recovery property in the second stage.

4.7.3 Advantages of BB-DPG

Both DPG and BB-DPG can reduce the read access and computation load in the second stage parity generation, and thus makes this process more efficient. This is particularly beneficial when this process needs to be completed quickly, either because of the short time window of system idling, or the process is triggered by certain urgent system event. In these cases, DPG and BB-DPG have significant advantages over reading all the data to generate the second group of parities.

The main differences between DPG and BB-DPG are as follows: firstly, in BB-DPG two codes $\mathcal{C}$ and $\bar{\mathcal{C}}$ are used as base codes, while in DPG, they are forced to be the same; secondly, data and parity interleaving is allowed in BB-DPG, while this was not allowed in DPG. The benefits of BB-DPG is that the two codes $\mathcal{C}$ and $\bar{\mathcal{C}}$ can provide us more flexibility in the bitmatrix design as discussed in the next section, and allowing interleaving is clearly beneficial for load balancing. It is also worth noting that DPG can be viewed as a special case of the piggy-backing framework

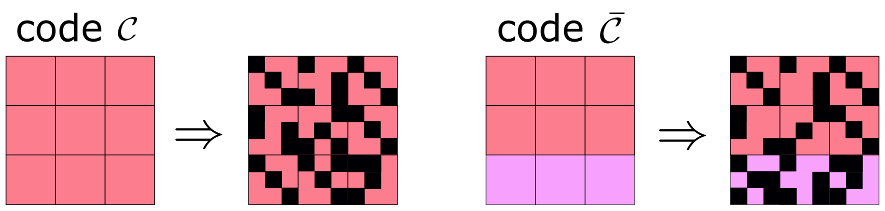

Figure 4.4: Optimizing bitmatrics when $(k, m.m') = (3, 2, 3)$. The parts of the parity coding matrices in light red are those involved in the first stage parity generation, and we wish to minimize the number of 1's in the corresponding bitmatrices.

[43], however BB-DPG no longer belongs to the piggy-backing code class since two different base codes $\mathcal{C}$ and $\bar{\mathcal{C}}$ are used, and interleaving is also introduced.

4.8 Bitmatrix-Based Delayed Parity Generation

With BB-DPG, we can now optimize the coding matrix in the two codes $\mathcal{C}$ and $\bar{\mathcal{C}}$ for various desired properties. Our focus will be for higher computation throughput in the first stage, since this may likely occur at high system-load time.

4.8.1 Optimizing Bitmatrices

Consider the coding matrix, the first stage encoding needs to compute the m' parities for the first m data stripe, and thus we wish to have the bitmatrix corresponding to the m'-by-k parity coding matrix of $\mathcal{C}$ to have few 1's. The first stage encoding also requires the computation of the first m parities of the last $m' - m$ stripes, and thus the bitmatrix corresponding to the m-by-k parity coding matrix of $\bar{\mathcal{C}}$ should also have few 1's. Thus we essentially wish to find two different coding matrices with converted bitmatrices optimized for different parts; see Fig. 4.4. It is seen that using BB-DPG allows better flexibility in this optimization compared to DPG, because here the two codes $\mathcal{C}$ and $\bar{\mathcal{C}}$ can be optimized separately for different criteria.

There are various methods for this optimization task, as summarized in [4], whose accompanying open source package is available at [58]. We utilize this package to optimize the coding matrices for $\mathcal{C}$ and $\bar{\mathcal{C}}$ in this work.

Table 4.3: Computation Performance Comparison in Schedule Length

(k, m, m')	BB-DPG	optimized DPG		DPG w/o opt		(k, m) code		(k, m') code	
(5,2,3)	198	201	1.52%	228	15.15%	138	-30.30%	216	9.09%
(6,2,4)	366	376	2.73%	408	11.48%	228	-37.70%	472	28.96%
(8,2,4)	506	516	1.98%	540	6.72%	324	-35.97%	656	29.64%
(9,2,4)	558	578	3.58%	600	7.53%	372	-33.33%	752	34.77%
(9,3,5)	999	1007	0.80%	1036	3.70%	690	-30.93%	1165	16.62%
(10,4,6)	1724	1732	0.46%	1780	3.25%	1260	-26.91%	1908	10.67%
(12,3,6)	2334	2343	0.39%	2523	8.10%	1512	-35.22%	3066	31.36%
(16,4,8)	5692	5772	1.41%	6044	6.18%	3680	-35.35%	7544	32.54%

4.8.2 Throughput Evaluation

The coding throughput of the BB-DPG is compared with other possible approaches in Table 4.4, we show the encoding schedule length of BB-DPG with optimized bitmatrics vs. 1) DPG with optimized bitmatrix, 2) DPG without an explicitly optimized bitmatrix and 3) no DPG applied ($(k, m), (k, m')$ code). in Table 4.3. The percentage indicates the difference of schedule length compared to BB-DPG.

Table 4.4 gives the coding throughput comparing to other codes. Here DPG using finite field based erasure coding library is also included. The percentage indicates the loss compared to BB-DPG. It can be seen that BB-DPG is able to perform better than directly using a matrix without explicit optimization. It is also able to perform better than DPG, mainly due to the flexibility in choosing the bitmatrices. The improvements are relatively modest. We suspect more advanced optimization procedures will lead to more significant improvements, however, in this work we utilized the off-the-shelf optimization procedure (genetic algorithm) in the software package [58].

In Table 4.4, we also compare the encoding throughput BB-DPG with the strategy of not using DPG at all. The last column gives the throughput for optimized (k, m') erasure code. Compared to BB-DPG, it requires computing more parities, and thus a loss in throughput is expected. The second last column gives the throughput for optimized (k, m) erasure code. Compared to BB-DPG, it essentially requires computing less parities, and thus a gain in throughput is expected. In general, the computation load of BB-DPG is between the two, and in a system with relatively

Table 4.4: Coding Throughput: BB-DPG vs. DPG vs. without DPG (in MB/s)

(k,m,m')	stage	BB-DPG	optimized DPG		DPG w/o opt		DPG in GF		(k,m) code		(k,m') code	
(5,2,3)	enc	5552	5428	-2.23%	5125	-7.69%	4818	-13.22%	7232	30.26%	5304	-4.47%
	dec1	6857	6572	-4.16%	6651	-3.00%	5974	-12.88%	7714	12.50%	-	
	gen	11978	12516	4.49%	10841	-9.49%	10151	-15.25%	-		-	
	dec2	5114	5144	0.59%	4991	-2.41%	4083	-20.16%	-		5302	3.68%
(6,2,4)	enc	5194	4969	-4.33%	4768	-8.20%	4195	-19.23%	6944	33.69%	4434	-14.63%
	dec1	6342	6577	3.71%	6477	2.13%	5740	-9.49%	7670	20.94%	-	
	gen	7048	7184	1.93%	6663	-5.46%	5684	-19.35%	-		-	
	dec2	4421	4482	1.38%	4383	-0.86%	3002	-32.10%	-		4517	2.17%
(8,2,4)	enc	5068	5145	1.52%	4915	-3.02%	4235	-16.44%	6997	38.06%	4341	-14.34%
	dec1	6869	6816	-0.77%	6875	0.09%	5900	-14.11%	7695	12.03%	-	
	gen	7316	7575	3.54%	7241	-1.03%	5781	-20.98%	-		-	
	dec2	4535	4517	-0.40%	4423	-2.47%	2956	-34.82%	-		4605	1.54%
(9,2,4)	enc	4910	5216	6.23%	5107	4.01%	4218	-14.09%	6969	41.93%	4293	-12.57%
	dec1	6801	6807	0.09%	6802	0.01%	5874	-13.63%	7698	13.19%	-	
	gen	7628	7600	-0.37%	7550	-1.02%	5612	-26.43%	-		-	
	dec2	4481	4399	-1.83%	4527	1.03%	2923	-34.77%	-		4522	0.91%
(9,3,5)	enc	4034	4139	2.60%	4031	-0.07%	3004	-25.53%	5375	33.24%	3637	-9.84%
	dec1	4908	4994	1.75%	4846	-1.26%	3385	-31.03%	5521	12.49%	-	
	gen	7751	7818	0.86%	6802	-12.24%	5607	-27.66%	-		-	
	dec2	3740	3789	1.31%	3705	-0.94%	2197	-41.26%	-		3825	2.27%
(10,4,6)	enc	3092	3069	-0.74%	3050	-1.36%	2208	-28.59%	4410	42.63%	2847	-7.92%
	dec1	3880	3884	0.10%	3881	0.03%	2640	-31.96%	4483	15.54%	-	
	gen	7531	7668	1.82%	7527	-0.05%	5249	-30.30%	-		-	
	dec2	3048	3053	0.16%	3037	-0.36%	1776	-41.73%	-		3045	-0.10%
(12,3,6)	enc	3433	3393	-1.17%	3280	-4.46%	2569	-25.17%	4719	37.46%	2712	-21.00%
	dec1	4150	4174	0.58%	4139	-0.27%	3133	-24.51%	4394	5.88%	-	
	gen	5086	5057	-0.57%	4891	-3.83%	3579	-29.63%	-		-	
	dec2	2704	2682	-0.81%	2661	-1.59%	1588	-41.27%	-		2666	-1.41%
(16,4,8)	enc	2721	2727	0.22%	2644	-2.83%	1857	-31.75%	3989	46.60%	2119	-22.12%
	dec1	3364	3363	-0.03%	3391	0.80%	2049	-39.09%	3516	4.52%	-	
	gen	4024	3934	-2.24%	3933	-2.26%	2604	-35.29%	-		-	
	dec2	2105	2067	-1.81%	2075	-1.43%	1023	-51.40%	-		2024	-3.85%

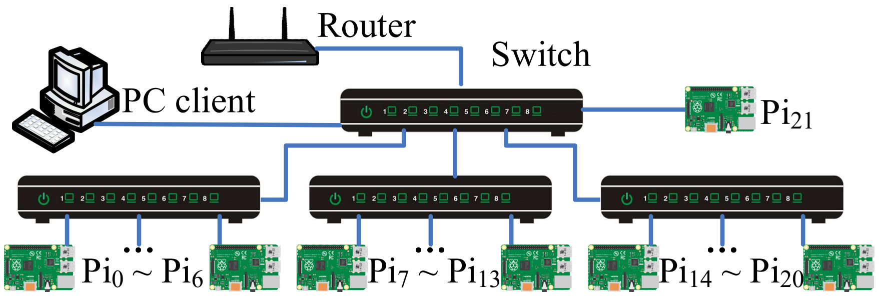

Figure 4.5: The local storage system based on Raspberry PIs.

abundant computation resource, BB-DPG does not cause a significant increase in terms of the computation load.

4.9 Prototype Systems and Experiments

We implemented a simple prototype system, to test and understand whether delayed parity generation can indeed improve the system write performance as expected.

4.9.1 System Setups

The storage software is multi-threaded, where a coding thread performs the coding task, and a transmission thread sends the data and parity blocks to different storage nodes. The single block that the transmission thread sends to each storage node is referred to as a *transmission unit*, or a TU. A detailed hardware specification report is given in Table 4.5. The data flow of these tests is RAM$\rightarrow$network$\rightarrow$RAM, between the client and the data nodes; no disk IOs are involved in these tests.

Our local testbed is based on a wirelined system with 22 Raspberry PIs, interconnected through gigabit network; see Fig. 4.5. The network is constructed with a tree topology to mimic the typical network connectivity in data centers, where the switches are gigabit ethernet switches. In the 22 Raspberry PIs', 5 of them are PI model B, 1 PI-2 model B, and 16 PI-3 model B. The client here is a linux workstation which generates the data to write, and the encoding and decoding computations are performed by the client. In other words, the PIs only serve as the storage nodes. Multiple storage nodes are allowed to reside in the same physical one, but in most cases this can be avoided.

Table 4.5: Test Platform Specs

Name	Model	# of Nodes	CPU	RAM	Bandwidth
pi	PI model B	5	BCM2835	512 MB	11.3 MB/s
	PI-2 model B	1	BCM2837	1 GB	
	PI-3 model B	16	BCM2837	1 GB	
	workstation	1	16x AMD Ryzen 1700X	16 GB	
aws1	t2.medium	1	2x E5-2686v4	4 GB	118.54 MB/s
	t2.micro	24	1x E5-2676v	1GB	
aws2	c2n.xlarge	7	4x Xeon Platinum 8124	10.5 GB	596.27 MB/s

Two different virtual machine based setups are also used (on Amazon AWS). The first setup simulates a low performance data center system, which has 24 single-core AWS t2.micro nodes as the storage nodes and one dual-core t2.medium node is used as the client. The network is accessed with private IPs, where the bandwidth is only indicated as "low to moderate". Iperf tests suggest $\sim$118 MB/s network bandwidth between any two nodes. The second setup simulates a high performance system, which has 7 quad-core c5n.xlarge nodes. The network bandwidth is specified by AWS as "up to 25Gb". Iperf tests suggest $\sim$600 MB/s network bandwidth using private IPs. One node is used as the client whereas each of the other six has four virtual storage nodes.

We first experimented on different TU size, and observed that with a sufficient large TU size, the prototype system can almost exhaust all network bandwidth, while with a smaller TU size, the system is overwhelmed by the frequent TC transmissions and the throughput suffers. The bandwidth utilization will reduce as the number of nodes grows, but only by $\leq 10\%$ even in our largest test ((16,4,8) code). In order to avoid the small TU size penalty, we choose 256 KB, 512 KB and 512 KB as the TU size of the three test systems, respectively (except for the stage of generating new parities, which is limited to $(m' - m) * w * packetsize$). An (5,2,3) system has been deployed to all three systems and the raw data transmission rate of different TU size is metered. As shown in Fig 4.6, with small TU size, the system is overwhelmed by the frequent TC transmissions and the throughput suffers.

Table 4.6: System Throughput: BB-DPG vs. without DPG (in MB/s) (pi)

(k, m, m')	stage	BB-DPG	(k, m) code		(k, m') code	
(5,2,3)	enc	7.87	7.86	-0.04%	6.88	-12.53%
	dec1	11.06	11.00	-0.52%		-
	gen	15.42		-		-
	dec2	11.01		-	11.01	-0.07%
(6,2,4)	enc	8.22	8.24	0.31%	6.59	-19.84%
	dec1	11.02	11.01	-0.10%		-
	gen	10.58		-		-
	dec2	11.01		-	11.01	-0.05%
(8,2,4)	enc	8.78	8.78	0.01%	7.31	-16.75%
	dec1	11.00	11.00	-0.02%		-
	gen	11.82		-		-
	dec2	10.96		-	10.96	0.02%
(9,2,4)	enc	8.94	8.99	0.57%	7.58	-15.24%
	dec1	10.96	10.98	0.23%		-
	gen	12.33		-		-
	dec2	10.93		-	10.91	-0.17%
(9,3,5)	enc	8.19	8.23	0.47%	7.04	-14.06%
	dec1	10.30	10.94	6.14%		-
	gen	13.43		-		-
	dec2	10.94		-	10.91	-0.34%
(10,4,6)	enc	7.79	7.87	0.93%	6.84	-12.19%
	dec1	10.90	10.96	0.51%		-
	gen	15.13		-		-
	dec2	10.92		-	10.92	-0.02%
(12,3,6)	enc	8.77	8.77	-0.02%	7.32	-16.60%
	dec1	10.95	10.94	-0.10%		-
	gen	11.74		-		-
	dec2	10.96		-	10.98	0.21%
(16,4,8)	enc	8.76	8.77	0.02%	7.32	-16.52%
	dec1	10.92	10.96	0.41%		-
	gen	11.76		-		-
	dec2	10.94		-	10.90	-0.38%

Table 4.7: System Throughput: BB-DPG vs. without DPG (in MB/s) (aws1)

(k, m, m')	stage	BB-DPG	(k, m) code		(k, m') code	
(5,2,3)	enc	74.45	74.35	-0.13%	61.74	-17.07%
	dec1	103.24	106.85	3.51%		-
	gen	29.36		-		-
	dec2	110.95		-	100.52	-9.40%
(6,2,4)	enc	71.22	77.83	9.28%	63.04	-11.48%
	dec1	101.15	29.44	-70.89%		-
	gen	33.13		-		-
	dec2	109.40		-	99.84	-8.74%
(8,2,4)	enc	81.98	80.13	-2.26%	65.95	-19.55%
	dec1	101.81	96.70	-5.02%		-
	gen	29.23		-		-
	dec2	108.70		-	94.30	-13.25%
(9,2,4)	enc	80.42	80.59	0.22%	67.66	-15.86%
	dec1	91.73	97.84	6.66%		-
	gen	28.10		-		-
	dec2	107.02		-	97.19	-9.18%
(9,3,5)	enc	71.18	76.73	7.80%	65.81	-7.54%
	dec1	8.99	29.15	224.10%		-
	gen	29.27		-		-
	dec2	96.32		-	96.70	-0.39%
(10,4,6)	enc	72.98	73.26	0.39%	44.63	-38.85%
	dec1	98.12	98.87	0.76%		-
	gen	25.61		-		-
	dec2	99.82		-	96.53	-3.29%
(12,3,6)	enc	54.46	80.21	47.28%	66.03	21.24%
	dec1	93.79	95.83	2.17%		-
	gen	29.99		-		-
	dec2	99.48		-	81.70	-17.88%
(16,4,8)	enc	74.30	77.68	4.55%	63.74	-14.21%
	dec1	89.71	92.27	2.85%		-
	gen	29.91		-		-
	dec2	98.24		-	89.69	-8.70%

Table 4.8: System Throughput: BB-DPG vs. without DPG (in MB/s) (aws2)

(k, m, m')	stage	BB-DPG	(k, m) code		(k, m') code	
(5,2,3)	enc	418.52	426.03	1.79%	368.76	-11.89%
	dec1	585.26	592.08	1.17%		-
	gen	614.81		-		-
	dec2	604.10		-	587.35	-2.77%
(6,2,4)	enc	416.14	440.36	5.82%	335.69	-19.33%
	dec1	553.10	595.18	7.61%		-
	gen	611.61		-		-
	dec2	565.71		-	554.02	-2.07%
(8,2,4)	enc	440.23	467.38	6.17%	353.81	-19.63%
	dec1	552.96	595.81	7.75%		-
	gen	700.06		-		-
	dec2	559.72		-	526.55	-5.93%
(9,2,4)	enc	418.44	443.54	6.00%	405.52	-3.09%
	dec1	584.47	556.65	-4.76%		-
	gen	695.96		-		-
	dec2	589.75		-	582.04	-1.31%
(9,3,5)	enc	399.33	437.52	9.56%	326.56	-18.22%
	dec1	537.24	596.72	11.07%		-
	gen	702.71		-		-
	dec2	539.57		-	505.01	-6.41%
(10,4,6)	enc	385.47	399.10	3.54%	330.66	-14.22%
	dec1	540.22	558.23	3.33%		-
	gen	673.61		-		-
	dec2	550.37		-	528.06	-4.05%
(12,3,6)	enc	455.40	406.26	-10.79%	380.94	-16.35%
	dec1	573.49	521.97	-8.98%		-
	gen	689.42		-		-
	dec2	584.45		-	573.27	-1.91%
(16,4,8)	enc	430.18	402.76	-6.37%	361.90	-15.87%
	dec1	545.55	529.48	-2.95%		-
	gen	635.83		-		-
	dec2	554.17		-	538.41	-2.84%

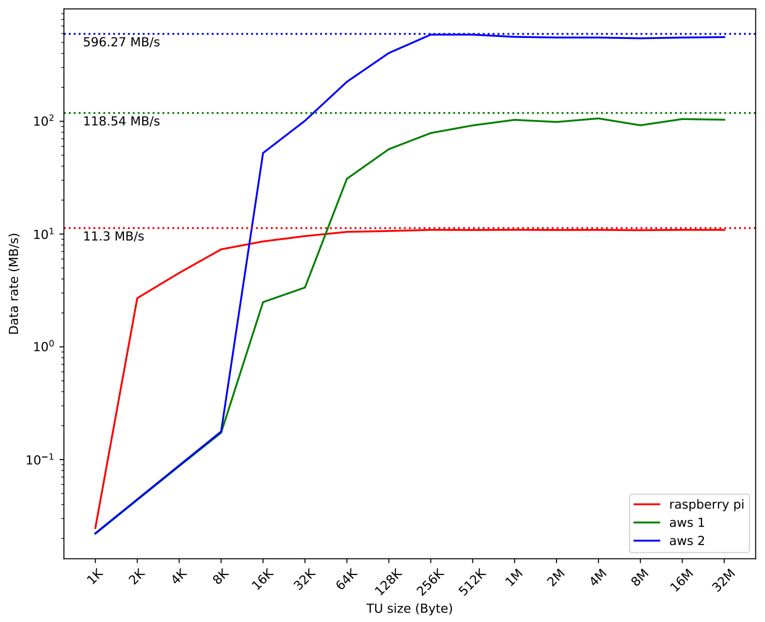

Figure 4.6: Network performance of 3 system for varying TU size. Dot line indicates the baseline network speed obtained using *iperf*

4.9.2 System Throughput Evaluation

We compare the average system performance with two erasure coded system with parameter (k, m) and that with (k, m'), respectively. Table 4.6 - 4.8 gives the quantitative result of overall throughput of all three systems and Fig. 4.7 focuses on the write throughput of all three systems, namely, pi, aws1, and aws2, from top to bottom. It is seen BB-DPG behaves similarly to a (k, m) code and has higher throughput than (k, m') code. Comparing the overall throughput in Table 4.6 - 4.8 and coding throughput in Table 4.4, it can be seen that all three systems are bandwidth-bounded.

We also observed that the performance variation is highly depended on the stability of the environment. Among the three systems, the pi-based system is the most stable option as there is no other traffic in the local network. Aws2 system is less stable and aws1 is the worst, which can be observed clearly for code (9,3,5) in Table 4.7. This is likely due to QoS throttle of these low-end VMs in a shared environment.

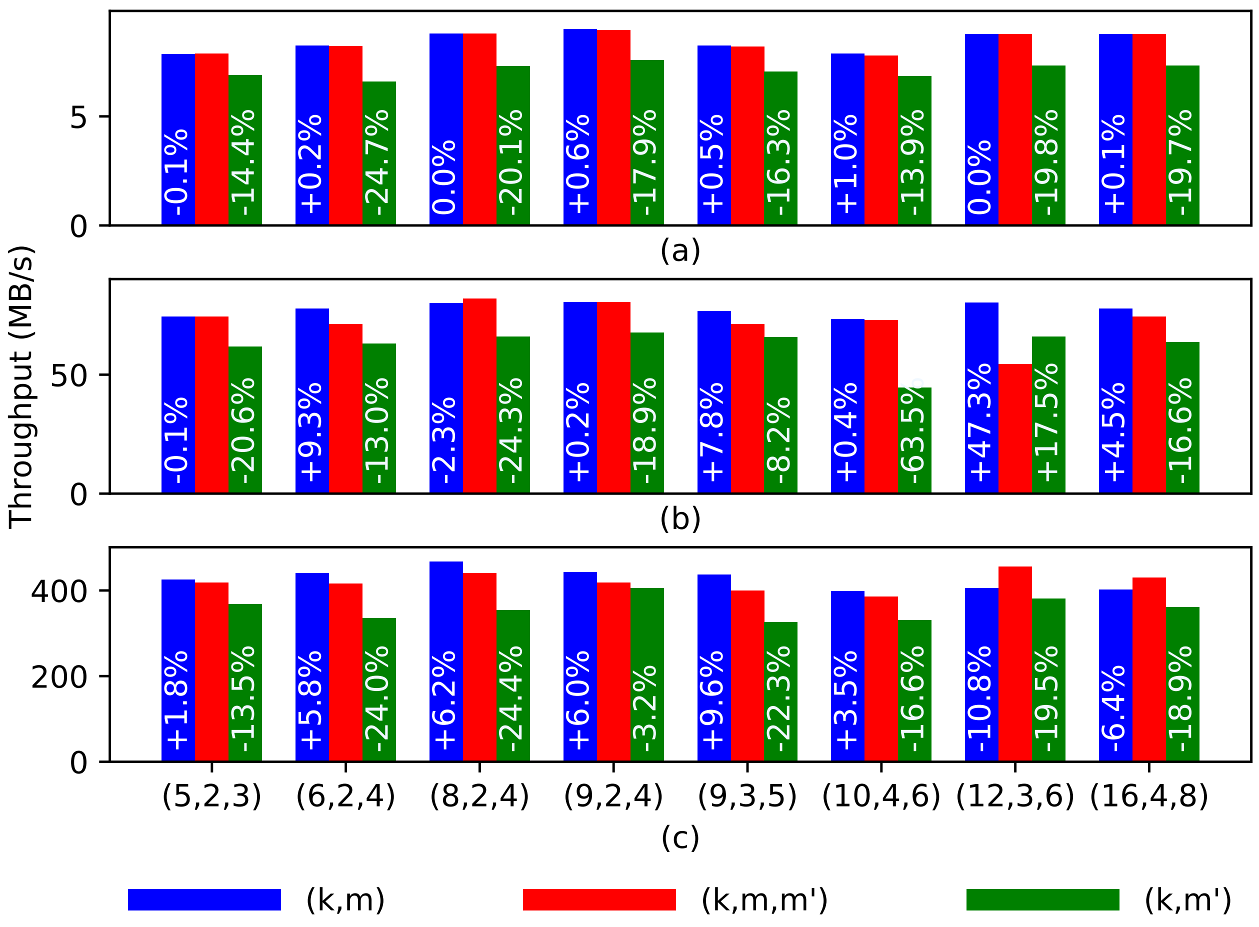

Figure 4.7: System write performance using (k, m) codes, BB-DPG, and (k, m') codes on (a) pi based system, (b) aws1: a low-end VM system and (c) aws2: a high-performance VM system.

4.10 Conclusion

We propose a delayed parity generation technique motivated by two scenarios in distributed data storage systems. This approach can be used to either improve the initial write (commit) speed, or provide better adaptivity in the storage system. Our result shows using MSR codes in either setting is unnecessary and the proposed code can accomplish the same functionalities with a simpler code, a smaller subpacketization factor, and at a lower computation level.

We then generalize the DPG code construction, which we refer to as BB-DPG. BB-DPG allows more flexibility in optimizing the coding bitmatrices. The encoding throughput showed that BB-DPG offers a modest computational advantage. A prototype system is constructed, and tested on a local platform and two virtual machine based platforms. The results confirm delayed parity generation can offer fast data write when the system is network (IO) bottlenecked.

5. SUMMARY AND CONCLUSION[1] [2]

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 flexibility and the better throughput performance, but also because of the simplicity in migration to new generation CPUs.

We also propose a delayed parity generation technique, which can be used in two scenarios in distributed data storage systems. By delaying the generation, transportation, and writing of some parities, 9this approach can be used to either improve the initial write (commit) speed, or provide better adaptivity in the storage system. Our result shows that blindly adopting MSR codes in either setting is unnecessary and wasteful, and the proposed code can accomplish the same functionalities with a simpler code, a smaller subpacketization factor, and at a lower computation level. The proposed codes can be further optimized in various ways. It should be noted that between the time that the initial data is written and the full set of parities are written, the system will be operating with a lower reliability than a system with the full set of parities. The effect of this difference in reliability in this short window can be quantified using existing models in the literature; see e.g.,

[1] ©2018 IEEE. Partly Reprinted, with permission, from S. Mousavi, T. Zhou and C. Tian, "Delayed parity generation in MDS storage codes", 2018 IEEE International Symposium on Information Theory (ISIT), June 2018

[2] ©2020 ACM. Partly Reprinted, with permission, from T. Zhou and C. Tian. 2020. "Fast Erasure Coding for Data Storage: A Comprehensive Study of the Acceleration Techniques". ACM Trans. Storage 16, 1, Article 7 (March 2020), 24 pages. DOI:https://doi.org/10.1145/3375554

[29].

A generalization of the DPG code construction is then provided, which we refer to as BB-DPG. BB-DPG allows more flexibility in optimizing the coding bitmatrices. The encoding throughput was evaluated, and the results showed that BB-DPG offers a modest computational advantage. We suspect the gain will be more pronounced with more powerful optimization pro- cedures. A prototype system is constructed, and tested on a local platform and two virtual machine based platforms. The results confirm delayed parity generation can offer fast data write when the system is network (IO) bandwidth constrained.

REFERENCES

[1] S. Mousavi, T. Zhou, and C. Tian, “Delayed parity generation in mds storage codes,” in *2018 IEEE International Symposium on Information Theory (ISIT)*, pp. 1889–1893, IEEE, 2018. ©2018 IEEE. Reprinted, with permission.

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

[3] T. Zhou and C. Tian, “Fast erasure coding for data storage: A comprehensive study of the acceleration techniques,” *ACM Trans. Storage*, vol. 16, Mar. 2020.

[4] T. Zhou and C. Tian, “Fast erasure coding for data storage: A comprehensive study of the acceleration techniques,” in *Proceedings. of 17th USENIX Conference on File and Storage Technologies (FAST 19)*, pp. 317–329, 2019.

[5] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson, “RAID: High-performance, reliable secondary storage,” *ACM Computing Surveys (CSUR)*, vol. 26, no. 2, pp. 145–185, 1994.

[6] Z. Wilcox-O’Hearn and B. Warner, “Tahoe–The least-authority filesystem,” in *Proceedings of the ACM Conference on Computer and Communications Security*, pp. 21–26, 2008.

[7] W. Litwin and T. Schwarz, “LH*RS: A high-availability scalable distributed data structure using Reed Solomon codes,” in *Proceedings of the ACM SIGMOD Record*, vol. 29, pp. 237–248, 2000.

[8] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, *et al.*, “Oceanstore: An architecture for global-scale persistent storage,” in *ACM SIGARCH Computer Architecture News*, vol. 28, pp. 190–201, ACM, 2000.

[9] K. D. Bowers, A. Juels, and A. Oprea, “Hail: A high-availability and integrity layer for cloud storage,” in *Proceedings of the 16th ACM conference on Computer and communications security*, pp. 187–198, ACM, 2009.

[10] I. S. Reed and G. Solomon, “Polynomial codes over certain finite fields,” *Journal of the society for industrial and applied mathematics*, vol. 8, no. 2, pp. 300–304, 1960.

[11] S. Lin and D. J. Costello, *Error control coding*. Pearson Education India, 2001.

[12] M. Ovsiannikov, S. Rus, D. Reeves, P. Sutter, S. Rao, and J. Kelly, “The Quantcast file system,” *Proceedings of the VLDB Endowment*, vol. 6, no. 11, pp. 1092–1101, 2013.

[13] H. Weatherspoon and J. D. Kubiatowicz, “Erasure coding vs. replication: A quantitative comparison,” in *In Proceedings of the First International Workshop on Peer-to-Peer Systems (IPTPS)*, 2002.

[14] J. S. Plank, “Tutorial: Erasure codes for storage applications,” in *Proc. of the 4th USENIX Conference on File and Storage Technologies*, pp. 1–74, 2005.

[15] S. Saalfeld, A. Cardona, V. Hartenstein, and P. Tomančák, “CATMAID: collaborative annotation toolkit for massive amounts of image data,” *Bioinformatics*, vol. 25, no. 15, pp. 1984–1986, 2009.

[16] J. Goecks, A. Nekrutenko, and J. Taylor, “Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences,” *Genome biology*, vol. 11, no. 8, p. R86, 2010.

[17] J. Dean and S. Ghemawat, “Mapreduce: simplified data processing on large clusters,” *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[18] “How cloud object storage works.” Available at https://www.ibm.com/cloud-computing/products/storage/object-storage/how-it-works/, Accessed: 2017-07-10.

[19] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, S. Yekhanin, *et al.*, "Erasure coding in Windows Azure storage.," in *Usenix Annual Technical Conference (ATC)*, pp. 15–26, Boston, MA, 2012.

[20] J. Arnold, *Openstack swift: Using, administering, and developing for Swift object storage*. O'Reilly Media, Inc., 2014.

[21] S. A. Weil, *Ceph: reliable, scalable, and high-performance distributed storage*. PhD thesis, University of California, Santa Cruz, 2007.

[22] H. Weatherspoon and J. D. Kubiatowicz, "Erasure coding vs. replication: A quantitative comparison," in *Proceedings of the International Workshop on Peer-to-Peer Systems*, pp. 328–337, 2002.

[23] M. Blaum, J. Brady, J. Bruck, and J. Menon, "EVENODD: An efficient scheme for tolerating double disk failures in RAID architectures," *IEEE Transactions on computers*, vol. 44, no. 2, pp. 192–202, 1995.

[24] P. Corbett, B. English, A. Goel, T. Grcanac, S. Kleiman, J. Leong, and S. Sankar, "Row-diagonal parity for double disk failure correction," in *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, pp. 1–14, 2004.

[25] C. Huang and L. Xu, "STAR: An efficient coding scheme for correcting triple storage node failures," *IEEE Transactions on Computers*, vol. 57, no. 7, pp. 889–901, 2008.

[26] C. Huang, J. Li, and M. Chen, "On optimizing XOR-based codes for fault-tolerant storage applications," in *Proceedings of Information Theory Workshop*, pp. 218–223, 2007.

[27] J. Luo, M. Shrestha, L. Xu, and J. S. Plank, "Efficient encoding schedules for XOR-based erasure codes," *IEEE Transactions on Computers*, vol. 63, no. 9, pp. 2259–2272, 2014.

[28] J. S. Plank, K. M. Greenan, and E. L. Miller, "Screaming fast Galois field arithmetic using intel SIMD instructions.," in *Proceedings of the 11st Usenix Conference on File and Storage Technologies*, pp. 299–306, 2013.

[29] J. Blomer, M. Kalfane, R. Karp, M. Karpinski, M. Luby, and D. Zuckerman, “An XOR-based erasure-resilient coding scheme,” Tech. Rep. TR-95-048, University of California at Berkeley, 1995.

[30] J. S. Plank, “The RAID-6 Liberation codes,” in *FAST-2008: 6th Usenix Conference on File and Storage Technologies*, (San Jose), pp. 97–110, February 2008.

[31] J. S. Plank, C. D. Schuman, and B. D. 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)*, pp. 1–12, 2012.

[32] J. S. Plank, “XOR’s, lower bounds and MDS codes for storage,” in *2011 IEEE Information Theory Workshop (ITW)*, pp. 503–507, 2011.

[33] “Intel® Intelligent Storage Acceleration Library,” 2016. Available at https://software.intel.com/en-us/articles/intel-isa-l-cryptographic-hashes-for-cloud-storage.

[34] R. Ahlswede, N. Cai, S. Y. R. Li, and R. W. Yeung, “Network information flow,” *IEEE Transactions on Information Theory*, vol. 46, pp. 1204–1216, Jul. 2000.

[35] S.-Y. Li, R. W. Yeung, and N. Cai, “Linear network coding,” *IEEE Transactions on Information Theory*, vol. 49, no. 2, pp. 371–381, 2003.

[36] S. Lin and D. J. Costello, *Error control coding*. Prentice Hall, 2 ed., 2004.

[37] J. Li, X. Tang, and C. Tian, “A generic transformation for optimal repair bandwidth and rebuilding access in MDS codes,” in *Proc. of 2017 IEEE International Symposium on Information Theory (ISIT)*, pp. 1623–1627, Jun. 2017.

[38] A. G. Dimakis, P. B. Godfrey, Y. Wu, M. Wainwright, and K. Ramchandran, “Network coding for distributed storage systems,” *IEEE Transactions on Information Theory*, vol. 56, pp. 4539–4551, Sep. 2010.

[39] V. Cadambe, S. Jafar, H. Maleki, K. Ramchandran, and C. Suh, “Asymptotic interference alignment for optimal repair of MDS codes in distributed storage,” *IEEE Transactions on Information Theory*, vol. 59, pp. 2974–2987, May 2013.

[40] N. B. Shah, K. V. Rashmi, P. V. Kumar, and K. Ramchandran, “Distributed storage codes with repair-by-transfer and non-achievability of interior points on the storage-bandwidth tradeoff,” *IEEE Transactions on Information Theory*, vol. 58, pp. 1837–1852, Mar. 2012.

[41] K. V. Rashmi, N. B. Shah, and P. V. Kumar, “Optimal exact-regenerating codes for distributed storage at the MSR and MBR points via a product-matrix construction,” *IEEE Transactions on Information Theory*, vol. 57, pp. 5227–5239, Aug. 2011.

[42] M. Ye and A. Barg, “Explicit constructions of high-rate MDS array codes with optimal repair bandwidth,” *IEEE Transactions on Information Theory*, vol. 63, pp. 2001–2014, Apr. 2017.

[43] K. Rashmi, N. B. Shah, and K. Ramchandran, “A piggybacking design framework for read- and download-efficient distributed storage codes,” *IEEE Transactions on Information Theory*, vol. 63, pp. 5802–5820, Sep. 2017.

[44] I. Tamo, Z. Wang, and J. Bruck, “Access versus bandwidth in codes for storage,” *IEEE Transactions on Information Theory*, vol. 60, no. 4, pp. 2028–2037, 2014.

[45] M. Ye and A. Barg, “Explicit constructions of optimal-access MDS codes with nearly optimal sub-packetization,” *IEEE Transactions on Information Theory*, vol. 63, pp. 6307–6317, Oct. 2017.

[46] Y. Hu, “The MDS scaling problem for cloud storage,” in *Presentation at First Workshop on Network Coding and Data Storage*, 2011.

[47] B. K. Rai, V. Dhoorjati, L. Saini, and A. K. Jha, “On adaptive distributed storage systems,” in *2015 IEEE International Symposium on Information Theory (ISIT)*, pp. 1482–1486, 2015.

[48] H. Zhang, H. Li, B. Zhu, X. Yang, and S.-Y. R. Li, “Minimum storage regenerating codes for scalable distributed storage,” *IEEE Access*, vol. 5, pp. 7149–7155, 2017.

[49] S. Goparaju, I. Tamo, and R. Calderbank, "An improved sub-packetization bound for minimum storage regenerating codes," *IEEE Transactions on Information Theory*, vol. 60, pp. 2770–2779, May 2014.

[50] N. B. Shah, K. V. Rashmi, P. V. Kumar, and K. Ramchandran, "Interference alignment in regenerating codes for distributed storage: necessity and code constructions," *IEEE Transactions on Information Theory*, vol. 58, pp. 2134–2158, Apr. 2012.

[51] F. MacWilliams and N. Sloane, *The Theory of Error-Correcting Codes*. North Holland Publishing Co., 1977.

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

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

[54] J. S. Plank, J. Luo, C. D. Schuman, L. Xu, and Z. Wilcox-O'Hearn, "A performance evaluation and examination of open-source erasure coding libraries for storage.," in *Fast*, vol. 9, pp. 253–265, 2009.

[55] J. S. Plank and C. Huang, "Tutorial: Erasure coding for storage applications." Slides presented at FAST-2013: 11th Usenix Conference on File and Storage Technologies, February 2013.

[56] S. Balaji and P. V. Kumar, "A tight lower bound on the sub-packetization level of optimal-access MSR and MDS codes," *arXiv preprint arXiv:1710.05876*, Oct. 2017.

[57] "Apache: HDFS erasure coding." Available at https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HDFSErasureCoding.html, Accessed: 2017-07-15.

[58] "Zerasure: Another open source library for erasure coding." https://github.com/zhoutl1106/zerasure, 2019.

APPENDIX A

PROOF OF BB-DPG

In this appendix, we prove that BB-DPG satisfies the following three requirements: 1) after the first stage it can withstand a loss of any m devices; 2) after the second stage it can withstand a loss of any m' devices; 3) the second stage parities can be built by accessing only the data and first stage parities in the last $m' - m$ stripes. We assume there are $k + m'$ storage devices, and allow different patterns of placement of coded symbols in different stripes.

Recover data after the first stage: Notice that regardless the interleaving, with a loss of any m nodes, each stripe has at least k symbols remaining. For the first m stripes, we can thus recover all the data, i.e., $d^{(1)}, d^{(2)}, \ldots, d^{(m)}$, and all the parties $p^{(1)}, p^{(2)}, \ldots, p^{(m)}$, by using the decoding and encoding functions of $\mathcal{C}$. In the last $m' - m$ stripes, if a parity symbol is not lost, then it is in the form $\bar{p}_i^{(m+j)} + p_{m+j}^{(i)}$ where $1 \leq j \leq m' - m$ and $1 \leq i \leq m$. Therefore, the component $p_{m+j}^{(i)}$ is a parity symbol in the first m stripes, and can be eliminated. As a consequence, for the last $m' - m$ stripes, say stripe $(m + j)$, we also have at least k symbols in the vector $(d_1^{(m+j)}, d_2^{(m+j)}, \ldots, d_k^{(m+j)}, \bar{p}_1^{(m+j)}, \bar{p}_2^{(m+j)}, \ldots, \bar{p}_m^{(m+j)})$. Since the code $\bar{\mathcal{C}}$ can recover the data from any k symbols, the data in the last $m' - m$ stripes can also be recovered.

Recover data after the second stage: Essentially the same argument as above applies also here, and we omit the details.

Delayed parity generation: From the data symbols of any tripe in the last $m' - m$ stripes, say, the $(m + j)$-th stripe, the parities $(\bar{p}_{m+1}^{(m+j)}, \bar{p}_{m+2}^{(m+j)}, \ldots, \bar{p}_{m'}^{(m+j)})$ can be generated from $d^{(m+j)}$ using the encoding functions of code $\bar{\mathcal{C}}$; these are the required second stage $(m' - m)$ parity symbols for stripe-$(m + j)$. Further notice that $\bar{p}_i^{(m+j)} + p_{m+j}^{(i)}$ was also read for $i = 1, 2, \ldots, m$ and $j = 1, 2, \ldots, m' - m$. However, the component parity $\bar{p}_i^{(m+j)}$ can also be computed from the $(m + j)$-th stripe's data symbols $d^{(m+j)}$. This guarantees that the last $m' - m$ parities of the first m stripes can also be generated.