A DELAYED PARITY GENERATION CODE FOR ACCELERATING DATA WRITE IN ERASURE CODED STORAGE SYSTEMS

A Thesis Presented for the
Master of Science
Degree
The University of Tennessee, Knoxville

Sara Mousavicheshmehkaboodi
December 2017

Copyright © 2017 by Sara Mousavicheshmehkaboodi
All rights reserved.

ABSTRACT

We propose delayed parity generation as a method to improve the write speed in erasure-coded storage systems. In the proposed approach, only some of the parities in the erasure codes are generated at the time of data write (data commit), and the other parities are not generated, transported, or written in the system until system load is lighter. This allows faster data write, at the expense of a small sacrifice in the reliability of the data during a short period between the time of the initial data write and when the full set of parities is produced. Although the delayed parity generation procedure is anticipated to be performed during time of light system load, it is still important to reduce data traffic and disk IO as much as possible when doing so. For this purpose, we first identify the fundamental limits of this approach through a connection to the well-known multicast network coding problem, then provide an explicit and low-complexity code construction. The problem we consider is closely related to the regenerating code problem. However, our proposed code is much simpler and has a much smaller subpacketization factor than regenerating codes. Our result shows that blindly adopting regenerating codes in this setting is unnecessary and wasteful. Experimental results confirm that to obtain the improved write speed, the proposed code does not significantly increase computation burden.

TABLE OF CONTENTS

1 Introduction **1**

1.1 Erasure Codes 2

1.2 Motivation and contribution of this work 4

1.2.1 Accelerating write speed 5

1.2.2 Scalability of erasure codes 6

2 Background and relevant concepts **8**

2.1 Finite Fields 8

2.1.1 Addition 8

2.1.2 Multiplication 9

2.2 Maximum distance separable codes 9

2.3 Regenerating codes 12

3 Delayed parity generation codes **15**

3.1 The system model 15

3.2 Data read requirements for DPG 17

3.3 Code construction 20

3.3.1 The Coding Procedure 21

3.3.2 MDS property of DPG 24

3.4 Reliability of erasure coded systems 24

4 Implementation **26**

4.1 Introduction to the implementation 26

4.1.1 Encoding 27

4.1.2 New parity generation 28

4.1.3 Decoding 30

5 Performance **32**

5.1 Memory and I/O cost 32

5.2 CPU cost 34

5.3 Comparison 35

6 Conclusion and Discussion **39**

List of References **40**

Appendix **46**

A The code **47**

A.1 Encoder 47

A.2 Delayed parity generation 57

A.3 Decoder 65

A.4 Functions and headers 75

A.4.1 Functions 75

A.4.2 Headers 81

A.5 Dependencies 82

Vita **82**

LIST OF TABLES

Table 2.1: Converting a polynomial to an equivalent one with an order less than n in $GF(2^n)$. 10

Table 4.1: The content of the metadata file. 29

LIST OF FIGURES

Figure 1.1: The structure of an erasure code: encodes the original data into k+m nodes through encoding process and recovers the failed nodes using the content of the remaining nodes through decoding. 2

Figure 1.2: A structure of how data is stored on nodes in a data center. . 3

Figure 3.1: 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 being read in the second stage, while A_i's are the parts that not being read. C_i's are the delay parity nodes. 16

Figure 3.2: The graphical model used in the proof of Theorem 3.1 18

Figure 3.3: Stacking of multiple MDS codes. 21

Figure 3.4: Systematic and parity symbols in the two stages. 22

Figure 5.1: Comparing the performance of RS and DPG code for different number of systematic nodes 36

Figure 5.2: Comparing the performance of RS and DPG code for different number of initially-generated parities. 37

Figure 5.3: Comparing the performance of RS and DPG code for different number of delayed-generated parities. 37

Figure 5.4: Comparing the performance of RS and DPG code for files with different sizes. 38

CHAPTER 1

INTRODUCTION

With the advent of big data, many other related areas such as storage techniques and devices, system performance, reliability, scalability and storage overhead have attracted many researchers' attentions. Companies such as Facebook, Google and Amazon are dealing with petabytes of data every single day [1]. Storing such an amount of data in a single storage device is not practical. This is due to the cost of the devices and also lack of reliability. Distributed storage systems (DSS) are the alternative option for storing large scale data. A distributed storage system could be in data centers such as GFS[2] or a peer-to-peer system such as Total recall[3] and OceanStore[4]. A data center is a collection of many clusters in which there are a large number of interconnected nodes. Using distributed storage systems provides scalability. However, it can also introduce some other problems. Scalability is provided because new nodes could be easily added to the system without being a threat to the performance of the system. The problems are initiated by the need for sharing data between nodes and managing the join/leave activity of the nodes.

Individual nodes are unreliable and in the danger of failing for different possible reasons. This can result in an unreliable system. In order to have reliability, we need to be able to repair or reconstruct nodes as they fail. One straightforward and commonly used method is keeping multiple copies of the data across different geographically located storage nodes. Storage systems that use this technique are referred to as replication-based storage systems. Original Google File System (GFS) [2], and HDFS [5] are examples of such systems. They store three replicas of each chunk of data. Looking at the amount of data that is stored or needs to be stored by such systems provides a very rational reasoning that replication-based schemes are not the most storage efficient methods for providing a given reliability. We need to use another approach to overcome this inefficiency. Redundant Array of Independent Disks (RAID), is a parity-based technology that could be used as a solution for the aforementioned inefficiency. Using RAID, some redundant data (called parity) is added to the original data to provide reliability. In the event of

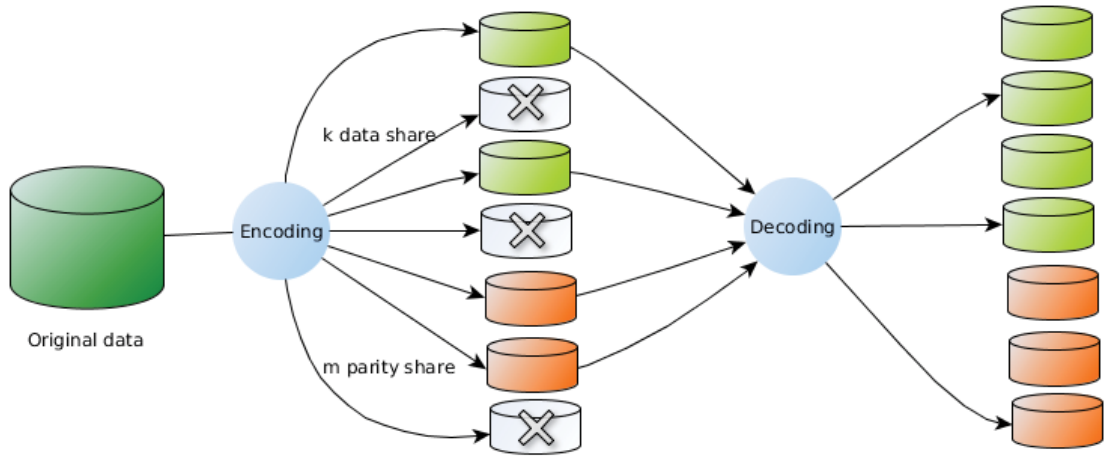


Figure 1.1: The structure of an erasure code: encodes the original data into k+m nodes through encoding process and recovers the failed nodes using the content of the remaining nodes through decoding.

node failure, these a combination of the available data and parities are used to recover the information in the failed node. There are different versions of RAID that uses XOR-based techniques and other coding methods to create the parities. Using RAID, one complete version of the input data along with the redundant portion are stored. The amount of redundant data, however, is less than the size of the original data. Thus, RAID is more storage efficient compared to replication methods. However, the drawback of it is that it is not able to tolerate more than two failures simultaneously (e.g. it can not recover the original data in the case of losing more than two nodes). That is when erasure codes come into play.

1.1 Erasure Codes

Erasure codes are methods for protecting data from being lost. Erasure codes add redundancy to the system to provide reliability while trying to reduce the storage overhead compared to that of replication-based methods. Mathematically, an erasure code is shown as (n, k), in which n is the total number of storage nodes, k of which are known as *systematic nodes* and $n - k = m$ are parity nodes where $n > m$.

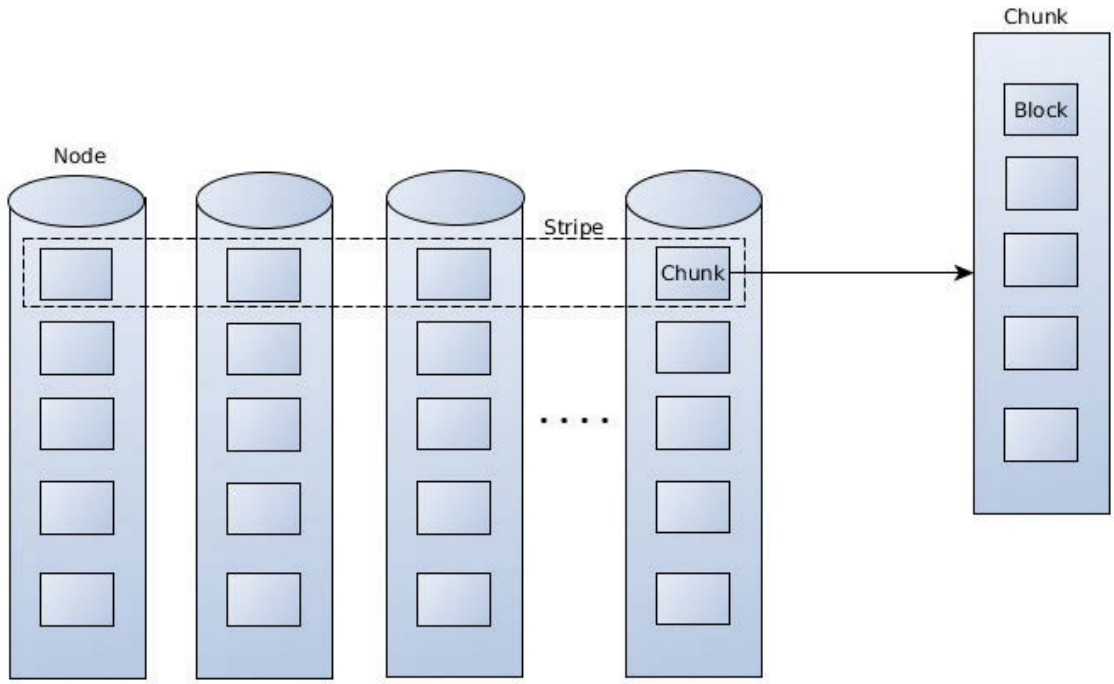


Figure 1.2: A structure of how data is stored on nodes in a data center.

Systematic nodes will contain the whole original file with no changes added (in an uncoded format)[6]. The parity nodes that are the redundant part of the data are created using the data from systematic nodes through a linear transformation. There are two main processes when using an erasure code: *encoding* and *decoding*. Encoding includes the process of generating the data that is supposed to be stored on the n nodes. Decoding, however, is regarded as recovering the lost data from the surviving nodes. The general concept is illustrated in figure 1.1.

In an erasure coded data storage system, when a file is going to be stored, it will be first partitioned into blocks with fixed sizes (usually 1MB [7]). Blocks are stored in groups named chunks (usually 64MB in GFS [2] and HDFS [7]). A node contains multiple chunks of data. One row of chunks across nodes (the part of data that is encoded and decoded together) is called a stripe. Data from a stripe across k nodes are used to generate $n - k$ parity symbols. Figure 1.2 illustrates the way that data is stored on nodes in a data center.

There are different metrics that affect the performance of an erasure coded data storage. These metrics need to be considered when deciding about parameters of the erasure code that is going to be used in a data center. These metrics are as follows:

- **Fault tolerance** is defined as the maximum number of node failures that the system can tolerate.
- **Storage overhead** is the amount of data that is needed to be stored in addition to the original data for having a given reliability.
- **Repair bandwidth** is defined as the amount of data that is required to be accessed and read for recovering the data in a failed node.
- **Complexity** is the amount of resources (e.g. time, CPU, and memory) that are needed for the computation in an erasure code.

Different performance metrics of erasure codes have been studied for years. Several designs such as [8], [3], and [9] use erasure codes instead of replication. An analysis by [10] shows that repair bandwidth and storage overhead in some cases can be reduced by an order of magnitude when using erasure codes compared to replication-based methods. Furthermore, [11], [12], and [13] are some efforts on erasure codes that provide optimal redundancy-reliability. Some other works such as [14], [15], [16], [17] are interested in low encoding, decoding and update complexity. These codes are mostly based on XOR operations. A tutorial by Plank [18] at USENIX FAST 2005 provides definitions of different erasure codes.

1.2 Motivation and contribution of this work

In this work, we focus on two main problems.

- Increasing the write speed of the process of storing data in an erasure coded data storage system
- Scalability or adaptivity of erasure coded data storage systems

1.2.1 Accelerating write speed

The capability for an erasure-coded data storage system to withstand a certain number of device failures comes from added redundancy when the data is written in the system. The data will remain intact if the number of failed devices is not greater than the number of parities. Thus, the larger the number of parity nodes, the more reliable the data. For example, well known RAID-6 based systems have two parities which can withstand two disk failures [19], the Quantcast QFS system utilizes a code with three parities which can withstand three node failures [20], and an option of HDFS-EC can use four parities [21] which allows four node failures. The parity data written in the system that is generated from the raw information data 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. The question here is whether the write speed can be improved in such erasure-coded data storage systems.

In this work, we propose a method based on *delayed parity generation* to improve the write speed (data commit speed), in erasure-coded data storage systems. We refer to our method as DPG in the rest of this work. The simple observation that motivated the proposed approach is that the target number of parities is usually chosen to guarantee the reliability of the data *over a long period of time*. However, if the data is protected using a smaller number of parity nodes *over a short period of time*, the sacrifice in the reliability is usually acceptable. As such, we can first write data into the system coded using an erasure code with a smaller number of parities, m, then during system idle or at light-load time, generate, transport, and write the additional parity data, $m'-m$, to reach the targeted long-term reliability. This process allows fast data write at the expense of a small reduction in the reliability during a short period of time between the initial data write (commit) and the delayed parity generation time. Although the delayed parity generation procedure is anticipated to be performed during time of light system load, it is still beneficial and important to reduce data traffic and disk IO as much as possible when doing so. This is because when accessing a node in a DSS, the node becomes unavailable to the network. However, availability is a very important property for a DSS. Thus, in order to provide high availability

and low traffic, one needs to keep IO overhead, node access and data read as low as possible.

1.2.2 Scalability of erasure codes

The problem that we mentioned in section 1.2.1 is also closely related to the so-called adaptivity (or scaling) of erasure-coded data storage systems [22, 23, 24, 25, 26, 27, 28]. Scalability of erasure codes was mentioned for the first time in the talk in "the MDS Scaling Problem for Cloud Storage" given by Y. Hu in the "First Workshop on Network Coding and Data Storage," [29]. 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. Engineers who are in charge of building large distributed data storage systems must choose proper hardware (e.g. consumer-grade vs. enterprise-grade, or hard drives vs. solid state disks), software architecture and various system parameters. However, the failure probabilities of storage devices and other system components, and the repair efficiency of the systems, which depend on the specific hardware components and system integration as well as software system architecture, can only be estimated accurately after the complete system has been deployed at scale. This means that a sufficiently large amount of data must have been stored in the system such that typical applications can utilize it. Thus the following issue arises: the intended reliability of the erasure codes, which needs to be chosen before the data is even written into the system, should be determined according to the failure probabilities and the behavior of the overall system. This can not be known accurately at the time of system deployment. Moreover, adapting the reliability of the erasure codes after data has been written can be costly if not done carefully. Adaptive erasure codes can help solve this issue.

When adjusting erasure code parameters, one may need to either decrease or increase the number of parities due to the demand for having different levels of reliability at different times. Decreasing parities is equivalent to code puncturing, which has been studied in [6]. However, increasing parities is much less straightforward. It can be seen that the delayed parity generation approach that we propose is equivalent to this problem when pre-planning at the time of data encoding is allowed. The code we propose can be used for this purpose, by first

encoding the data and generating the first sub-set of parities, and then when needed, generate the additional parities with a minimum amount of data read compared to existing solutions.

CHAPTER 2

BACKGROUND AND RELEVANT CONCEPTS

We mentioned some general information about erasure codes in the previous chapter. In this chapter we will provide more details about erasure codes and related concepts. We will first have a brief introduction to finite fields, then we will explain minimum distance separable codes. Finally, details and definitions about regenerating codes will be provided.

2.1 Finite Fields

All the arithmetic operations required in erasure codes are done in a finite field. As the name implies, a finite field is a finite set of elements in which addition, subtraction, division and multiplication are defined, such that some specific rules hold. The number of elements of a finite field is called the *order*. Since subtraction is adding a negative number and division is multiplying by the reciprocal of the number, we often hear a finite field as a set $\mathbb{F}$ of elements with two operations (addition and multiplication). A finite field of order q exists if and only if order q is equal to p^n, where p is a prime number and n is a positive integer. The notation for that is $GF(p^n)$. In the case that n equals 1, the calculation is very simple. One only needs to do the regular operation modulo p. That, however, is not the case when n is not 1. Erasure code's arithmetic operations are mostly done in $GF(2^n)$ where n is usually a power of 2 such as $\{4, 8, 16, 32, 64, 128\}$.

2.1.1 Addition

Each number can be represented in different ways such as binary, decimal and polynomial. For instance, if we consider number 10, a binary representation would be 1010 while the polynomial representation is $x^3 + x$. In order to do addition in

$GF(2^n)$ we only need to sum up the coefficients of the polynomial representations of the operands modulo 2. This means the addition in $GF(2^n)$ is equivalent to bitwise XOR. For example, $10 + 9$ in $GF(2^4) = (x^3 + x) + (x^3 + 1) = x + 1$. This is equal to $1010 \oplus 1001 = 0011 = x + 1$

2.1.2 Multiplication

The multiplication operation is not as simple as addition. For multiplication, we need to use the polynomial representation of the operands as well. We first do a regular multiplication and then will see if the order of the result is less than n or not. If the resulted polynomial, p, has a lower order than n, then the calculation is completed. However, if p has a higher order than n, then we need to convert it into another equivalent polynomial such that the order becomes less than n. This is done using an irreducible polynomial by calculating the polynomial modulo the irreducible polynomial. An irreducible polynomial over a field is a polynomial that its coefficients are members of that field and it can not be shown as the product of two other non-constant polynomials. Let's assume the order of p is a. In order to reduce a and convert p into another polynomial with a smaller order than a, we need to XOR the p with $x^{a-n} * (\textit{the irreducible polynomail})$ and repeat the process until a becomes less than n. For instance, for multiplying 2 by 4 in $GF(2^4)$ we will have $x * (x^2) = x^3 = 8 = p$. The order of p is 3 which is less than 4 and therefore 8 is the correct result. However, for multiplying 11 by 12 we will have $(x^3 + x + 1) * (x^3 + x^2) = x^6 + x^5 + x^4 + x^2$ which the order, 6, is greater than n and we should reduce it. To do so, we use the irreducible polynomial $x^4 + x + 1 = 10011$ for $GF(2^4)$. Table 2.1 shows the steps taken to reduce the order of the resultant polynomial. Therefore $11 * 12 = 13$.

2.2 Maximum distance separable codes

In a (n,k) erasure code C, the raw data ,B finite field symbols, is written as a row vector $\vec{d}$ of length k, each entry of which is in a certain finite field $\mathbb{F}_q$. The redundant portion of the data (parities) is generated using the content of $\vec{d}$ via an encoding

Table 2.1: Converting a polynomial to an equivalent one with an order less than n in $GF(2^n)$.

Polynomial	order	n	n-order	$x^{n-order} * (irreducable\ polynomial)$
1110100 $\oplus$1001100	$a = 6$	$n = 4$	$a - n = 2$	$x^2(x^4 + x + 1) = x^6 + x^3 + x^2 = 1001100$
111000 $\oplus$100110	$a = 5$	$n = 4$	$a - n = 1$	$x(x^4 + x + 1) = x^5 + x^2 + x = 100110$
11110 $\oplus$10011	$a = 4$	$n = 4$	$a - n = 0$	$x^4 + x + 1 = 10011$
1101	$a = 3$	$n = 4$		

process. In the case that $B = k$, C is a scalar code. The result is written in vector $\vec{p}$ of length m, each entry of which in the same finite field as the raw data. 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 $\mathbb{F}_q$. The left k by k submatrix of G is an identity matrix which results in having the original data in an uncoded shape in the first k nodes (known as systematic nodes). The product can be written as the concatenation of two vectors $[\vec{d}, \vec{p}]$, where vector $\vec{p}$ of length m is referred to as the parities. Mathematically, this can be expressed as

$$[\vec{d}, \vec{p}] = \vec{d} \cdot G = \vec{d} \cdot [I_k, P]. \tag{2.1}$$

Let's name the first k symbols $d_0, d_1, \cdots, d_{k-1}$ and the remaining $n - k = m$ parity nodes as $p_0, p_1, \cdots, p_{m-1}$. As opposed to the data nodes that contain intact and uncoded data, parity nodes contain coded data. The coded data is a linear combination of the nodes containing the original data as shown in Eq. 2.2

$$\begin{aligned} p_0 &= g_{k,0}d_0 + \cdots + g_{k,k-1}d_{k-1} \\ p_1 &= g_{k+1,0}d_0 + \cdots + g_{k+1,k-1}d_{k-1} \\ &\vdots \\ p_{m-1} &= g_{k+m-1,0}d_0 + \cdots + g_{k+m-1,k-1}d_{k-1} \end{aligned} \tag{2.2}$$

where $g_{i,j}$ are the elements of matrix G. Each element in the resulted vectors is called a codeword. The maximum Hamming distance between any two code-words in C is

referred to as $d(C)$ and it is upper bounded by the Singleton bound [30]

$$d(C) \leq n - k + 1. \tag{2.3}$$

Any $d(C) - 1$ loss of data in this code structure can be recovered using the remaining data. Thus, when the symbols in $[\vec{d}, \vec{p}]$ are distributed to different storage devices (nodes) $d(C) - 1$ device failures will not cause any data loss.

It is shown that if the finite field $\mathbb{F}$ is large enough, using Read Solomon structure [31] a code C with the maximum $d(C)$ (i.e. $d(C) = n - k + 1$) can be constructed. The resulted code is called *Maximum Distance Separable(MDS)* code. Since $d(C) = n - k + 1$ in MDS codes, they can tolerate up to $n - k = m$ failures.

In order to achieve a code with such a distance, matrix G needs to be chosen such that the submatrix of any k columns of G is full rank. Using full rank coefficient matrices provides the ability of recovering data through a matrix inversion operation when it's needed. There are many different methods for choosing matrix P, such as those based on Vandermonde matrices [18] or based on Cauchy matrices [30].

The most well known property of MDS codes is being able to recover the whole original data by accessing the content of any k nodes. Therefore, MDS codes provide a high reliability for a given storage overhead. However, MDS codes such as RS codes are inefficient in terms of repair bandwidth in the case of a single failure. This is because one needs to download B amount of data for repairing the content of a single lost node, in order to first reconstruct the original data and later repair the failed nodes. In other words, for repairing a single node, we need to download the same amount of data as we need for reconstructing the whole data. Therefore, the access rate (the number of nodes that are accessed in the repair process), for MDS codes is k and the repair bandwidth, γ, is as follows.

$$\gamma = k \times \frac{B}{k}. \tag{2.4}$$

The high rate of repair bandwidth of MDS codes was the motivation of research for codes that could relatively solve or improve this inefficiency. For example, regenerating codes are more focused on reducing repair bandwidth in distributed erasure coded storage systems and are described in the next section.

2.3 Regenerating codes

The main focus of regenerating codes is to reduce the amount of data read and transfer in the recovery process. This class of erasure codes are based on MDS codes with the modification of the number of symbols stored in each node. This means that each node can store multiple symbols. The number of finite field symbols that each node needs to store is called the subpacketization factor associated with the code. Regenerating codes should also be able to withstand loss of any m nodes in order to achieve the best storage efficiency.

In such codes, the data, B, is divided into k data shares, each of size $\alpha = B/k$, and m parity shares, each with the same size as each data share. Parity shares are generated from the k data shares. The integer value α is the subpacketization factor.

While repairing a failed node using regenerating coded data storage, β amount of data is downloaded from d surviving nodes, such that $\alpha \leq d\beta < B$, $d \geq k$. The recovered node may or may not contain the same content as it previously had. In the case of having the same content, the repair process is called *Exact repair* and otherwise it is called *Functional* repair [23]. When using functional repair, the resulting code should still maintain the property of being able to reconstruct the original data by accessing any k nodes.

Choosing the right numbers for parameters α and β is important. There is a trade-off between α and β that are the subject of *Minimum Storage Regenerating Codes* and *Minimum Bandwidth Regenerating Codes*.

Minimum storage regenerating (MSR) codes are regenerating codes that prioritize storage efficiency over optimal repair property. This means they minimize the amount of data stored in each node first and then reduce the repair bandwidth. According to [22], minimum storage in the case of a single failure is achieved if

$$(\alpha_{MSR}, \gamma_{MSR}) = (\frac{B}{k}, \frac{Bd}{k(d-k+1)}). \tag{2.5}$$

In other words, $\frac{\alpha}{d-k+1}$ symbols are downloaded from each node in order to repair a failure. From Eq. 2.5, γ is a decreasing function of d and the biggest d (e.g. $n-1$) leads to the minimum transfered data from each node.

Minimum bandwidth regenerating (MBR) codes are also a class of regenerating codes. However, MBR codes minimize repair bandwidth rather than storage overhead. In an MBR code, the capacity of each node equals the amount of data that is needed for the recovery of a failure (e.g. $\alpha = \gamma$). According to [22], minimum repair bandwidth in the case of a single failure is achieved for the following pair.

$$(\alpha_{MBR}, \gamma_{MBR}) = (\frac{2Bd}{2kd - k^2 + k}, \frac{2Bd}{2kd - k^2 + k}) \tag{2.6}$$

It has been stated in [22] that due to the fact that MBR codes add $\frac{2n-2}{2n-k-1}$ redundancy to each node, they are no longer optimal with respect to the provided rate of reliability for a given redundancy.

Much research has been done on regenerating codes [22, 23, 24, 25, 26, 28, 27]. Some examples that specifically have dealt with the cases with multiple failures are [23, 32, 33]. In such codes, any node failure can be repaired efficiently by downloading the minimum amount of data from the remaining nodes. In particular, a special class of codes can also minimize the amount of data read (data access) [28]. Such codes are designed to allow more efficient node repairs in data storage systems, since less data traffic and less disk IO imply more efficient repair of failed nodes. Regenerating codes are usually not scalar, and the subpacketization factor is usually large. This particularly the case for high-rate codes. Where rate of a code is defined as $\frac{k}{n}$ (larger number of parities results in a lower rate code).

The minimum amount of total data read (and data download) when repairing r failed nodes ($r \leq m'$) in a $(k+m', k)$ regenerating code system, when all $k+m'-r$ remaining nodes are allowed to participate in the repair, is given in [22, 23] which is equal to

$$\hat{B}_r = \frac{r(m'+k-r)}{km'}. \tag{2.7}$$

It is shown in [28] that for regenerating codes with optimal disk-IO, the subpacketization factor is bounded as

$$\alpha \geq m'^{\frac{k}{m'}}. \tag{2.8}$$

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

CHAPTER 3

DELAYED PARITY GENERATION CODES

In this section, we provide an explicit code construction for DPG with low complexity. The construction is based on a strategic modification of any maximum distance separable code (such as the Reed-Solomon code) [6].

It can be seen that regenerating codes can be used for delayed parity generation, by viewing the node repairing process as a delayed parity generation process. However, regenerating codes are in fact unnecessary and wasteful in this case. More precisely, in regenerating codes, the requirement is that the system can regenerate *any* failed nodes using the minimum amount of data traffic from the surviving ones. That is, however, under the limitation that the number of failures is less than a certain threshold. On the other hand, 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 traffic from the existing ones. Therefore, regenerating codes are not the best options for our motivated use-case.

As for the scalability and adaptivity of erasure codes, previous efforts [29, 34, 35] took the approach of directly applying regenerating codes. For the same reason as we have just explained, this is also rather unnecessary and wasteful.

In this chapter, first the system model is briefly described. Next, a calculation of how much data read is needed in order to have delayed parity generation is presented. Later, a code construction for the proposed method is provided. The reliability aspect of the method is covered at the end of this chapter.

3.1 The system model

Delayed parity generation, as the name implies, is done in two stages. In the first stage, an input file with size B is read and it is partitioned onto k nodes. Each node

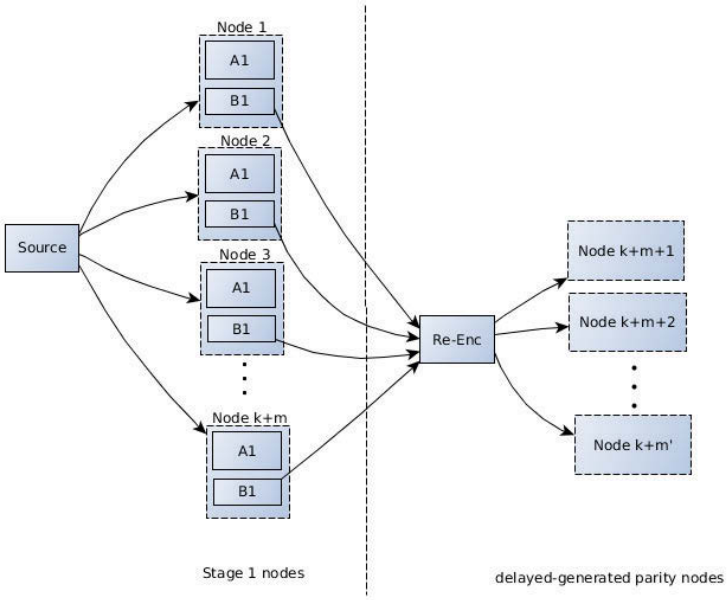


Figure 3.1: System model for delayed parity generation. The node labeled "Re-Enc" corresponds to the processing center in charge of the delayed parity generation. B_i's stand for the parts being read in the second stage, while A_i's are the parts that not being read. C_i's are the delay parity nodes.

will contain $\frac{B}{k}$ amount of data. This is the systematic portion of data that is used for generating the first m parity nodes. In this step we expect the resulted $(k+m, k)$ code to have the MDS property. In the second step, β amount of data is read from each of $k+m$ existing nodes. The downloaded data will be used for generating the last $m'-m$ parities. The output $(k+m', k)$ code should possess the MDS property as well. To put it all together, for creating a $(k+m', k)$ MDS code the first m out of m' parities are generated in the first stage and the $m'-m$ remaining ones are generated in the second stage. The two-stage encoding process is illustrated in figure 3.1, and since it is a function of parameters k, m, and m' we refer to it as (k, m, m') delayed parity generation.

3.2 Data read requirements for DPG

The performance of any valid coding strategy is measured by the total amount of data read B_r. According to the second stage of the model in section 3.1, $B_r \triangleq (k+m)\beta$. Since the total amount of initial data is B, and there is a linear relation between B and B_r, we can equivalently consider the per-unit data read $\bar{B}_r = \frac{B_r}{B}$. Allowing different amounts of data reads from nodes in the second stage cannot improve the optimal value of $\bar{B}_r$, and thus the uniform-read-amount model in section 3.1 is without loss of optimality in terms of $\bar{B}_r$. This can be shown through symmetrizing any code with non-uniform amounts of data, by way of space-sharing different placement patterns of the data shares and parity shares.

In this work, the focus is on the amount of *data read* B_r, instead of the amount of *data download* from the existing nodes. The amount of data read is more strict. It can be seen from the construction given in section 3.3.1 that the two measures reduce to the same. That means an optimal code allows simply transmitting from each existing node what is being read, without any need for further computation before transmission. This is beneficial in terms of system implementation, since complex computation is eliminated at the existing nodes in the second stage.

In this section we will determine the amount of data read that is required for DPG. Due to the connection of this work with multicast network coding, we got inspiration from [22] in terms of amount of data read. However, the problem setting in this work is different than that in [22] and we need to find a new bound.

Theorem 3.1. (k, m, m') *delayed parity generation can be accomplished if and only if*

$$\bar{B}_r \geq \min\left\{1, \frac{(m'-m)(k+m)}{km'}\right\}. \tag{3.1}$$

Proof. Let's consider a graphical representation of the problem as illustrated in figure. 3.2. There are $(m+k)$ type A nodes. They represent the parts in the $(m+k)$ coded shares produced in the first stage that are not read during the second stage. Each of these nodes has a size of $\gamma = B/k - \beta$. The $(m+k)$ type B nodes, each of which with capacity of β, represent the parts in the original $(m+k)$ coded shares that are

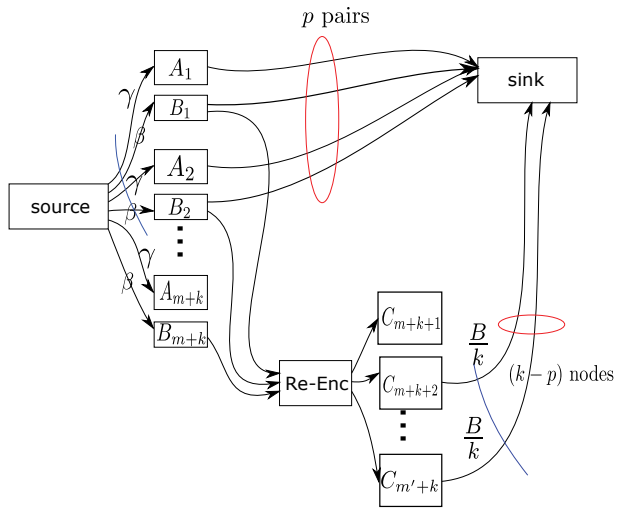


Figure 3.2: The graphical model used in the proof of Theorem 3.1

read during the second stage. The $(m' - m)$ type C nodes are the new parity shares, each with the capacity of B/k, generated in the second stage. Because of the coding requirements in the two stages as stated in the section 3.1, we can view this system as a multicast network. There are multiple possible sinks (a node that only receives data). 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

$$\sum_{p=\max(0,k-(m'-m))}^{k} \binom{k+m}{p} \binom{m'-m}{k-p} \tag{3.2}$$

different sinks in this multicast network. It follows from the well known network coding multicast result [36, 37] 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

$$\begin{aligned} B &\leq \min_p \left\{ p\frac{B}{k} + \min\left[(k-p)\frac{B}{k}, (k+m-p)\beta\right]\right\} \\ &= \min\left\{B, \min_p \left[p\frac{B}{k} + (k+m-p)\beta\right]\right\} \\ &= \min\left\{B, (k+m)\beta + \min_p p\left[\frac{B}{k} - \beta\right]\right\} \\ &= \min\left\{B, (k+m)\beta + (k-m'+m)\left[\frac{B}{k} - \beta\right]\right\}, \end{aligned} \tag{3.3}$$

where the outer minimization over p in Eq. 3.3 is to take into account all the possible sinks linked through different combinations of (A, B) pairs and C nodes. The inner minimization in Eq. 3.3 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

$$(k+m)\beta + (k-m'+m)\left[\frac{B}{k} - \beta\right] \geq B, \tag{3.4}$$

which further simplifies to

$$B_r = (k+m)\beta \geq \frac{(m'-m)(k+m)B}{km'}. \tag{3.5}$$

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

$$B_r = (k+m)\beta \geq B. \tag{3.6}$$

Combining the two cases in Eq. 3.5 and Eq. 3.6 together and normalizing both sides gives the desired bound.

The graph representation used above does not distinguish between data shares and parity shares, however it is well known that any linear code satisfying the conditions

can be converted to a code with clearly defined data shares and parity shares [6] through a linear transformation. This completes the proof of the theorem. □

As mentioned earlier, regenerating codes can be used for delayed parity generation although they have functions more than what are required in this context. It is worth noting that in this case, the bound on $\bar{B}_r$ in Theorem 3.1 indeed coincides with the corresponding minimum repair bandwidth derived for regenerating codes in [22] and [23]. This can be seen by setting $r = m' - m$ in [18]. Nevertheless, the fact that the per-unit data read in delayed parity generation has the same optimal value as the per-unit repair bandwidth in regenerating codes does not necessarily mean using regenerating codes for delayed parity generation is a wise choice. This will become more clear after we present an explicit code construction in the following section.

3.3 Code construction

The fundamental limits on the amounts of data read for delayed parity generation is provided by Theorem 3.1. However, we did not provide explicit code constructions. It is clear that code constructions for the case when

$$m' \geq k + m \tag{3.7}$$

is straightforward, which corresponds to the degenerate case of $\bar{B}_r = 1$, 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[1]. This strategy does not yield 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. In other words, 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.

[1]This is equivalent to a punctured code [6].

For the case when Eq. 3.7 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. These partial shares do not represent the complete stored data and thus the simple strategy for the degenerate case of Eq. 3.7 does not apply. In next section, we focus on this case and present a novel and low-complexity code construction for delayed parity generation.

3.3.1 The Coding Procedure

Let $G = [I_k, P_{k\times m'}]$ be the generator matrix of an arbitrary scalar systematic MDS code $\mathcal{C}$ with parameter $(k + m', k)$; for example, P can be chosen to be a k by m' Cauchy matrix or a Vandermonde matrix [18], [30]. 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.

Conceptually, we first partition the B information symbols into groups of k each (a total of m' groups), then encode each group using the matrix G; the i-th group after this encoding is thus the vector

$$[d_1^{(i)}, ..., d_k^{(i)}, p_1^{(i)}, ..., p_{m'}^{(i)}]; \tag{3.8}$$

where the first part is the information symbols $[d_1^{(i)}, ..., d_k^{(i)}]$, and the remaining part is the parity symbols. All of the m' groups together can be collected and represented as in Table 3.3, with each column corresponding to a vector in Eq. 3.8.

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'	$p_{m'}^{(1)}$	$p_{m'}^{(2)}$	$\cdots$	$p_{m'}^{(m')}$

Figure 3.3: Stacking of multiple MDS codes.

The parity symbols of the proposed code to be generated in the two stages are given in Table 3.4. It can be seen that in the first stage, the parity symbols in the first m columns are simply the original parity symbols of the $(k+m', k)$ MDS code $\mathcal{C}$. On the other hand, each entry of the other $m'-m$ parity symbols of these parity nodes are a linear 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 thus 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 the previous two steps.

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 lower left corner of Table 3.4, and the computation of the additions in step 3. There is no change in the alphabet size of code C, and thus the computation overhead is rather minimal.

	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	$d_k^{(1)}$	$d_k^{(2)}$	$\cdots$	$d_2^{(m)}$	$d_k^{(m+1)}$	$d_k^{(m+2)}$	$\cdots$	$d_k^{(m')}$
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'	$p_{m'}^{(1)}$	$p_{m'}^{(2)}$	$\cdots$	$p_{m'}^{(m)}$	$p_{m'}^{(m+1)}$	$p_{m'}^{(m+2)}$	$\cdots$	$p_{m'}^{(m')}$

Figure 3.4: Systematic and parity symbols in the two stages.

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

1. Read data $d_i^{(j)}$ for $i = 1, 2, \ldots, k$ and $j = m+1, m+2, \ldots, m'$ (the last $m'-m$ columns of the systematic data).

2. Read parities $p_i^{(j)} + p_j^{(i)}$ for $i = 1, 2, \ldots, m$ and $j = m+1, m+2, \ldots, m'$ (the last $m' - m$ columns of the parity data).

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 G 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 3.4. The amount of data read in the second stage is clearly given by $B_r = (m' - m)(k + m)$ with $\beta = (m' - m)$, and since $B = km'$, the per-unit data read is

$$\bar{B}_r = \frac{(m' - m)(k + m)}{km'}, \tag{3.9}$$

which matches the lower bound in Theorem 3.1 for the case $m' \leq k + m$.

It is important to note that the subpacketization factor for this code is m', while, if one uses optimal regenerating codes with the same amount of data reads, the subpacketization factor is at least $m'^{k/m'}$. This difference can be significant, for example, when $(k, m') = (10, 4)$ (a popular choice in HDFS-EC [21]), the proposed code has a subpacketization factor of 4. On the other hand, the solution based on regenerating codes will have a subpacketization factor of at least $4^{2.5} = 32$. Except for a much lower subpacketization factor, the proposed code has much lower computation complexity. Regarding using the presented code for adaptivity of erasure codes, it is practical to use DPG as long as we are allowed to plan ahead for the future changes on the number of parities. We note that although the existing efforts [29, 34, 35] are all on the case with pre-planning, in practice, it is more convenient if pre-planning is not used. That is, at the time of the data being initially written, we use an arbitrary erasure code without knowing whether or how much adaptivity is needed. Such case has not been considered before, and it is a potential future work.

3.3.2 MDS property of DPG

We need to show that the expectations mentioned in 3.1 (having MDS property for the resultant codes from stage 1 and 2) can be satisfied. This is equivalent to the following proposition.

Proposition 3.1.1. *The code given in Table 3.4 is a $(k+m', k)$ MDS code, when the code $\mathcal{C}$ is a $(k+m', k)$ MDS code.*

Proof. We show that any k out of the $k+m'$ rows in Table 3.4 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 $t+t_1+t_2 = k$. Let's denote the set of the available nodes as $\mathcal{A}$. It is clear that with this k rows of data as specified in Table 3.4, all the data in the first m columns can be recovered from the symbols in the rows of $\mathcal{A}$. This is because the code $\mathcal{C}$ is an MDS code and has the property that any k out of $k+m'$ symbols can be used to recover the data. For the rest of the columns, we need to generate $p_i^{(j)}$, $i = m+1, m+2, \ldots, m'$ and $j = 1, 2, \ldots, m$ by encoding the reconstructed $d_i^{(j)}$, $i = 1, 2, \ldots, k$ and $j = 1, 2, \ldots, m$. 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 3.3 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 3.3 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. □

3.4 Reliability of erasure coded systems

The reliability of data storage systems is usually measured by the mean time to data loss (MTTDL) [38, 39, 40, 41]. The more acceptable model for erasure-coded systems is given by Angusl [39] which assumes node failure and repair are Poisson, with failure rate λ and repair rate μ, in a k out of $(k+m)$ system. The MTTDL is given on the left hand side of the following equation, and can be approximated [41]

as the right hand side

$$\frac{\mu^m}{k\binom{m+k}{k}\lambda^{m+1}}\sum_{i=0}^{m}\binom{m+k}{i}\frac{\lambda^i}{\mu^i} \overset{\lambda \ll \mu}{\approx} \frac{\mu^m}{\lambda k\binom{m+k}{k}\lambda^m}. \tag{3.10}$$

Let us consider an example case. For typical device failure rate λ, e.g., 1 failure every 3 years, and repair rate μ, e.g., 6 per day, it can be shown that MTTDL is on the order of $5.4 * 10^{10}$ years with $(k, m) = (6, 4)$, and $8.5645 * 10^4$ years with $(k, m) = (6, 2)$. Thus with the latter, the probability of data loss within a 12-hour period is roughly $1.5995 * 10^{-8}$, which is rather small. As a comparison, the probability of data loss with $(k, m) = (6, 4)$ within a year is $1.8519 * 10^{-11}$. Therefore, if we use an erasure code of $(k, m) = (6, 2)$ during the 12-hour period, before the two more parities are fully generated, there is a small sacrifice in terms of the reliability before the data is fully committed with all parities properly generated. In exchange, we obtain improvement to the write speed at the time of data write, by reducing the total amount of data traffic and disk IO $2/(6 + 4) = 20\%$ in this example. Of course, the coding parameters and the delay period can be adjusted according to system requirements if the reliability requirement is more or less stringent, as well as adaptively with the system load variation.

CHAPTER 4

IMPLEMENTATION

Performing computations in *Finite Fields* adds complexity to the encoding and decoding process in an erasure-coded data storage system and requires more resources. Multiplications in finite fields can be very expensive in terms of CPU. This is one of the initial problems with RS codes. However, Intel's Streaming SIMD Extensions (SSE) [42] provides the opportunity of utilizing vectorization in these calculations. A well known library that benefits from this feature and provides fast finite field computations is gf-complete [43]. We used this library for our calculations. In addition, as mentioned in Chapter 3 we used a base MDS code for constructing DPG. Another open source library that we used in this work is Jerasure [44]. Jerasure includes an implementation of Reed Solomon codes that we utilize in DPG. Our implementation is done in the C language. The rest of this chapter covers the algorithms used for implementing DPG.

4.1 Introduction to the implementation

Our implementation contains three main functions. First is encoding. In the encoding function, the input file is encoded and stored onto $k + m$ nodes. This is the step in which only the first m parity nodes are generated. Next is the parity generation process, referred to as Re-Enc in Figure 3.1 of Chapter 3, in which the last $m' - m$ parities are generated and stored. The final step is the decoding process. The decoding phase is used to reconstruct the original file in the case of having one or up to m' failure(s). The implementation of these three steps are described respectively in the following sub-sections.

4.1.1 Encoding

As it can be inferred from Section 3.3.1, and more specificly from Table 3.4, for generating the first m nodes we need to have $p_i^j, j \in [0, m')$, where $i \leq m$ and $p_i^j, j \in [0, m)$ where $i > m$.

After generating the required parities, p_i^j for which $i \leq m$ and $j \in [m, m')$ need to be XORed with the parities located in their symmetric locations with respect to the prime diagonal. For example, p_1^2 will be XORed with p_2^1. Algorithm 1 shows the overall encoding process.

Algorithm 1 DPG's Encoding Algorithm

```
initialize parameters based on the inputs from the user;
pad_file();
set_buffer_Size();
encoding_number = ⌈fileSize / bufferSize⌉;
for i = 0 → i < encodingNumber do
    for j = 0 → j < m' do
        read_data();
        if j < m then
            MDS_encoder_m'();
        else
            MDS_encoder_m();
        end if
    end for
    for j = 0 → j < m do
        for z = m → j < m' do
            add( p_j^z, p_z^j);
        end for
    end for
    store();
end for
```

In the initialization step, all the parameters are initialized based on the inputs from the user. k is the number of systematic nodes, m is the initial number of parities, $m' - m$ is the number of parities that will be later added. Buffer size and an input file are also parameters that can be chosen by users. Buffer size can be set to 0 or any other number. If it is set to 0, then the buffer size will be automatically set in the program.

To automatically calculate the buffer size, the pad_file function increments the size of the input file such that $file_size \ \% \ k \times m' \times w = 0$. The variable new_file_size represents the incremented file size. The $encoding_number$ determines how many times the encoding process needs to be repeated. It is set to $\frac{new_file_size}{buffer_size}$ where $buffer_size$ is initially set to new_file_size by the program. After setting the $encoding_number$, $buffer_size$ is then set to $\frac{1}{m'} \times buffer_size$. This means that one full buffer will have as much data as that of the systematic data of one column from Table 3.3. The size of d_i^j equals $\frac{1}{k} \times buffer_size$, which is equal to the size of one block.

If the provided buffer size by the user is other than zero, it is incremented so that it is divisible by $k \times w \times m'$ and the pad_file function increases the file size such that $file_size \ \% \ (buffer_size) = 0$. The $encoding_number$ is set as $\frac{new_file_size}{buffer_size}$. Then $buffer_size$ is set to $\frac{1}{m'} \times buffer_size$ as before.

Having the parameters all set, we then generate the required MDS parities. The $MDS_encoder_m'$ and the $MDS_encoder_m$ function generate m' and m parities for the corresponding columns respectively. At this point, we add parity j from column i to parity i from column j where $i \in [0, m)$ and $j \in [m, m')$ (as in Table 3.4). This is equivalent to Step 3 from Stage one in Section 3.3.1. As it was mentioned earlier we perform all the calculation in $GF(2^w)$ and therefore, addition is equivalent to XOR. For XOR, $galois_region_xor$ from the gf-complete library is used. Regarding the generator matrix G mentioned in Section 2.2, we used a Vandermonde matrix in this implementation.

4.1.2 New parity generation

In this section we explain the implementation of creating the new $m' - m$ parities. In order to generate them we first download the contents of the right $m' - m$ columns of the $k + m$ existing nodes which is equivalent to Steps 1 and 2 in Stage 2 of Section 3.3.1. Next is pursuing Steps 3 and 4 from the same stage. Generating the new parities is shown in Algorithm 2.

Generating new parities starts with reading the required data which is the last $m' - m$ columns. When encoding, some information that will be needed for parity

Algorithm 2 DPG's New Parity Generation

```
1: Read the needed data;
2: for r = 0 → r < encodingNumber do
3:     for i = 0 → i < m' do //column
4:         for j = 0 → j < m' − m do //parity
5:             if i < m then
6:                 MDS_encode_1();//generates parities p_i^{j+m}
7:                 add();
8:             else
9:                 MDS_encode_1();
10:            end if
11:        end for
12:    end for
13: end for
14: store();
```

generation and decoding is saved as the metadata of the result in a text file. This information includes $encoding_number$, $buffer_size$, the original file size and so on.

Table 4.1: The content of the metadata file.

Variables	Description
$file_name$	The name of the input file
size	The size of the data stored in one column
k	Number of systematic nodes
m	Number of initially-generated parities
m'-m	Number of delayed-generated parities
w	Arithmetics are done over w-bit words
$buffer_size$	The size of the buffer
$vandermonde$	The type of encoding matrix that is used
$encoding_number$	Number of times that encoding needs to be done
$original_size$	Size of the input file before padding

In our construction, the first $(m' - m)m$ new parities are generated from solving the equations in the downloaded p_i^j $(i \in [0, m), j \in [m, m'))$ and the $(m'-m)(m'-m)$ remaining ones are generated from encoding the downloaded systematic data d_i^j $(i \in [0, m), j \in [m, m'))$. For solving the equations, we need the parities p_i^j to be generated from the downloaded systematic data. Generating these parities and then XORing them with the downloaded parities gives us the first $(m' - m)m$

new parities. In Algorithm 2, this is done through $MDS_encode_1()$ and $add()$. $MDS_encode_1()$ only generates one single parity. After solving the equations, the last $m' - m$ columns will have the data that MDS code C needs to reconstruct the systematic data corresponding to those columns. This is what we have in the *else* section of Algorithm 2.

4.1.3 Decoding

Decoding process is used when some of the n nodes generated from encoding an input file provided by a user are failed and one wants to reconstruct the original data using the remaining portion of the data. Decoding involves reading the remaining content (at least k surviving nodes out of $k + m'$) to recover the original data. The downloaded equations from parity nodes, $p_i^j, i \in [0, m), j \in [m, m')$, need to be solved. Based on what we have in Section 3.3.1, the parities in the first m columns are not combined with other parities. Therefore, any k subpackets that are left from these columns is what is needed by MDS code C to reconstruct the symmetric data corresponding to those columns. By reconstructing this data, we are able to generate the parities that are needed for solving the equations $p_i^j, i \in [0, m), j \in [m, m')$. XORing the generated parities with the downloaded data from $p_i^j, i \in [0, m), j \in [m, m')$, we will have what is needed for reconstructing the lost data for the last $m' - m$ columns using MDS code C.

We keep track of the failed nodes and the remaining ones so that we know what rows of encoder matrix G are going to be used in the decoding process. Algorithm 3 shows the steps for reconstructing the original file from the k remaining nodes.

The C code for the described functions are provided in the appendix.

Algorithm 3 DPG's Decoder Algorithm

```
Read the data from k surviving nodes;
for r = 0 → r < encodingNumber do
    for i = 0 → i < m do
        read_remaining_data();
        keep_track_of_lost_ones();
        reconstruction_using_single_symbols();
        regenerating_required_parities();
        for j = m → j < m′ do //column
            solve_equation();
        end for
    end for
    for i = m → i < m′ do
        reconstruction_using_single_symbols();
    end for
end for
```

CHAPTER 5

PERFORMANCE

In this section the performance of DPG in terms of I/O, memory and CPU compared to a basic MDS code, Read Solomon, is provided.

5.1 Memory and I/O cost

In order to analyze the memory cost, we check the size of the encoding matrices used for RS and DPG. Furthermore, we are interested to know how much of the input file will be loaded into memory given a specific buffer size when using Reed Solomon code compared to that of DPG for encoding, new-parity-generation and decoding.

Let's first focus on these performance metrics for the encoding process. Regarding the encoding matrix, DPG and Reed Solomon code both use the same encoding matrix. The encoding matrix in RS code is a function of:

- Number of systematic nodes
- Total number of nodes (i.e. systematic + parity nodes = n)
- The size of each element. For example in the case of using $GF(2^w)$, each element is a w-bit word

Thus, the size of encoding matrix in RS and DPG codes is $n * k * w$.

As for the amount of data read from the input file during the first stage (generating the first m parity nodes), it is a function of buffer size. Here, by buffer size we mean the size of the buffer that is used in the RS code. Although we use the same buffer size for both RS and DPG, we later set the buffer size in the DPG to $\frac{1}{m'} \times buffer_size$ and instead of doing the encoding process once (as done in RS), we repeat the process m' times. The implemented RS code that we are comparing DPG with, reads a buffer size amount of data and encodes it into $k + m$ symbols and later stores them onto

$k+m$ nodes. Therefore, either the encoding process is done at once, which results in having the buffer size be at least as big as B (if B is the size of input file), or it is done in multiple times, in which case we will have a smaller buffer size but overall B amount of data is read. Considering the amount of read data over the input file size, $\frac{B}{B} = 1$ is read.

The same notion is true about the write rate. Overall, the amount of written data is the same for the two codes. However, at each step, the number of writes in DPG is m' times of RS code's but in smaller chunks. The amount of written data with respect to the raw data in RS is $\frac{k+m}{k}$ and in DPG is $\frac{(k+m)m'}{km'}$ which is equal to $1 + \frac{m}{k}$ for the both codes.

In the decoding process, DPG and RS codes have the same amount of data read because in both cases k nodes are left.

As for the new-parity-generation-process RS code needs to read in one buffer size of data - the systematic portion of a stripe. Then, it performs the regular encoding process. However, instead of generating m' parities, it generates only the last $m' - m$ parities. Thus, the amount of data read in new-parity-generation-process in RS codes is a buffer size. This means it reads $\frac{B}{B} = 1$, while with DPG, the first $m(m' - m)$ symbols are not generated through RS-encoding. They are generated through solving equations in the downloaded parities. The amount of data that is required by DPG to be in memory is $\frac{m'-m}{m'} \times buffer_size + m(m'-m) \times \frac{buffer_size}{km'} = \frac{buffer_size(m'-m)(k+m)}{km'}$. For the cases that $m'(k+m) > km'$, it is more efficient to read the whole B. Otherwise, we use the approach mentioned in 3.3.1. Thus, the amount of read over the input file size is

$$\bar{B}_r \geq \min\left\{1, \frac{(m'-m)(k+m)}{km'}\right\}. \tag{5.1}$$

In this section, we explain the improvement in the write speed and the amount of data read using DPG. In a typical network setting, where nodes are connected through wirelines, the transmission and write time using different codes (in the first stage) is proportional to the amount of data to be transmitted and written. Similarly, in the second stage, the time is proportional to the amount of data transmission and disk-IO. As such, we can directly use these measures for different system parameters

to understand the improvement in the write speed and delayed parity generation, in comparison with a naive approach. In fact, although the system is determined by three parameters (k, m, m'), these performance measures are functions of the relative redundancy in the two stages

$$\bar{m} = \frac{m}{k}, \quad \bar{m}' = \frac{m'}{k}. \tag{5.2}$$

The amount of reduction in the data write in the first stage is simply $\bar{m}' - \bar{m}$ (in terms of the amount of redundancy with respect to the raw information). The improvement over the approach of writing the full set of parities is thus $(\bar{m}' - \bar{m})/(1 + \bar{m}')$. The larger the difference, the more acceleration in data write we can obtain. The amount of improvement can be constrained as a function of the amount of targeted redundancy $\bar{m}'$. For high rate codes where $\bar{m}' \leq 1$, it can be seen that the maximum reduction in data traffic and disk IO is 50% in the first stage. For low rate codes, i.e., large $\bar{m}'$, the improvement can be even larger. For example, when $\bar{m}' = 3$ and $\bar{m} = 0.333$, the amount of reduction is 77.78%.

The amount of data read, in terms of $\bar{m}$ and $\bar{m}'$, Eq. 3.1 can be rewritten as

$$\bar{B}_r \geq \min\left\{1, \frac{(\bar{m}' - \bar{m})(1 + \bar{m})}{\bar{m}'}\right\}. \tag{5.3}$$

It can be seen that when $\bar{m}$ and $\bar{m}'$ are close, the saving in data read using the proposed approach is the most significant. The case when $\bar{m}'$ and $\bar{m}$ are very different, corresponds to the case when too many parities are left to be generated in the second stage. Thus, $\bar{B}_r = 1$ and all the data need to be read.

5.2 CPU cost

In this section we compare the complexity of the RS codes with our proposed code. We first show the complexity of the encoding and decoding process and later the complexity of delayed-parity-generation. For encoding, a $(k + m, k)$ RS code usually is based on multiplying a $(k+m) \times k$ encoding matrix by a vector of the raw data with length k. Thus, $m'(k + m)(k + k)$ operations are performed. In DPG the additional

operations that are done in the $m(m'-m)$ parities generated in the third step of the first stage mentioned in Section 3.3.1 and the process of generating the last $m'-m$ parities for the first m columns are in addition to the operations done in RS codes. Therefore, the number of operations is $m'(k+m)(k+k)+m(m'-m)(k+k)+m(m'-m)$.

In the decoding process, RS codes perform $m'k(k+k)$ operations which are the result of multiplying the inverse of the coefficient matrix by the content of the k remaining nodes where the rest of the nodes $(n-k)$ have failed, while in DPG the number of performed operations in the decoding process is $m'k(k+k)+m(m'-m)(k+k)+m(m'-m)$.

As for the new parity generation process, RS codes perform the regular encoding process. However, instead of generating m' parities, it generates only the last $m'-m$ parities. Thus, the complexity is $((k+k)m'(m'-m))$, while with DPG the first $m(m'-m)$ symbols are not generated through RS-encoding of the corresponding systematic data. They are calculated through XORing the downloaded parity data with the first m parities of the last $m'-m$ columns. Therefore, the exact complexity would be $(k+k)m'(m'-m)+m(m'-m)$. The additional $m(m'-m)$ is for XOR operations required for generating the last $m'-m$ parities correspond to the first m columns.

The experimental results for running an RS code and DPG for different parameters are provided in Section 5.3.

5.3 Comparison

For the sake of comparison, both DPG and RS codes have been tested on one of the machines in UTK's Hydra lab. Each Hydra machine is an Intel Core i7-6700 at 3.40 GHz with 16GB memory.

In this section, a basic RS code and DPG are tested with different parameters to see how they perform for different parameter settings. Plots that are provided in this section are for a different number of systematic nodes (k), initially generated parities (m), delayed generated parities ($m'-m$) and file sizes.

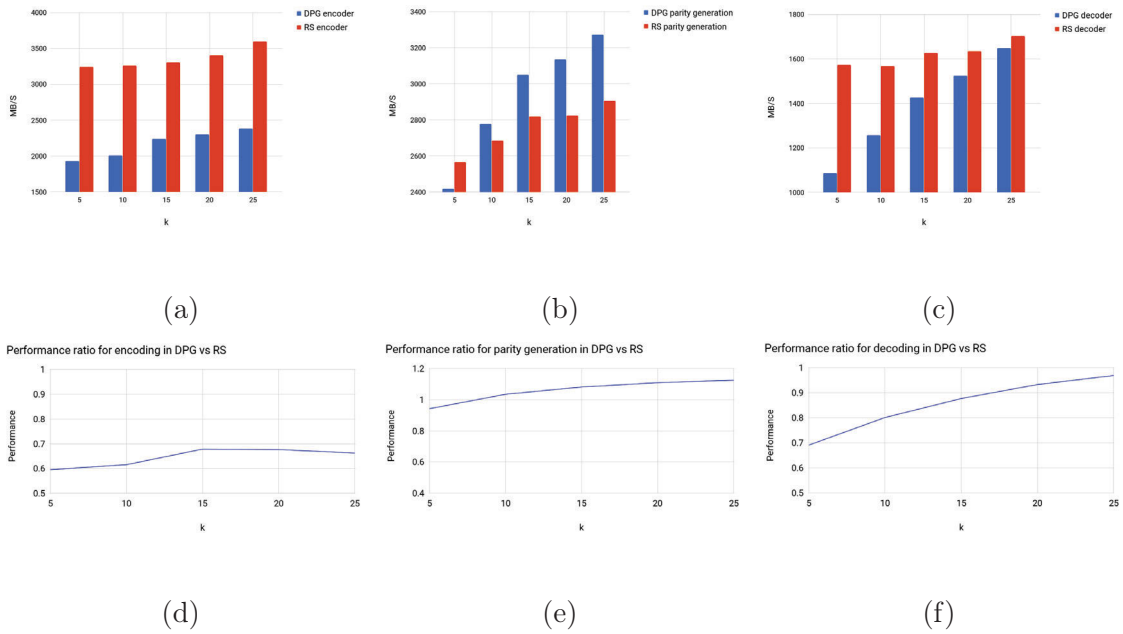


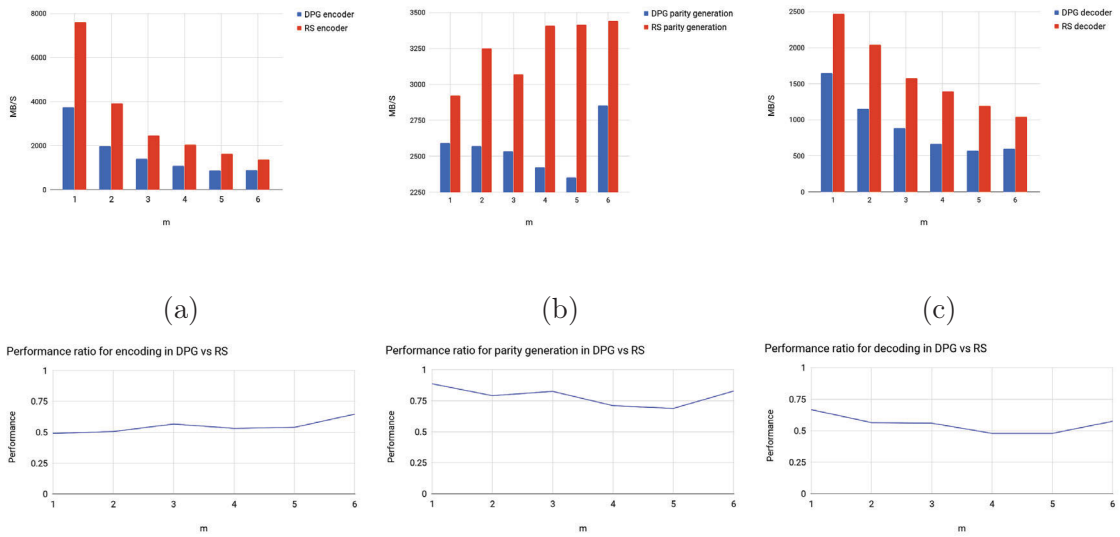
Figure 5.1: Comparing the performance of RS and DPG code for different number of systematic nodes

Figure 5.1 shows the performance result of DPG and RS for a different number of systematic nodes. In Figure 5.1 (a), (b) and (c) show the performance of DPG and RS code for encoding, parity generation and decoding respectively. In this figure, (d), (e) and (f) show the ratio of the performance of DPG and RS for encoding, delayed parity generation and decoding respectively. In Figure 5.1, (d), (e) and (f) correspond to (a), (b) and (c), respectively.

Figure 5.2 shows the result of running the RS code and DPG for different number of parities $\{3, 4, 5, 6, 7, 8\}$, where the number of initially generated parities (m) are $\{1, 2, 3, 4, 5, 6\}$, and the number of delayed-generated parities $(m' - m)$ is 2.

In Figure 5.3, the RS code and DPG are run for different numbers of delayed-generated parities $(m' - m)$ while m is 2.

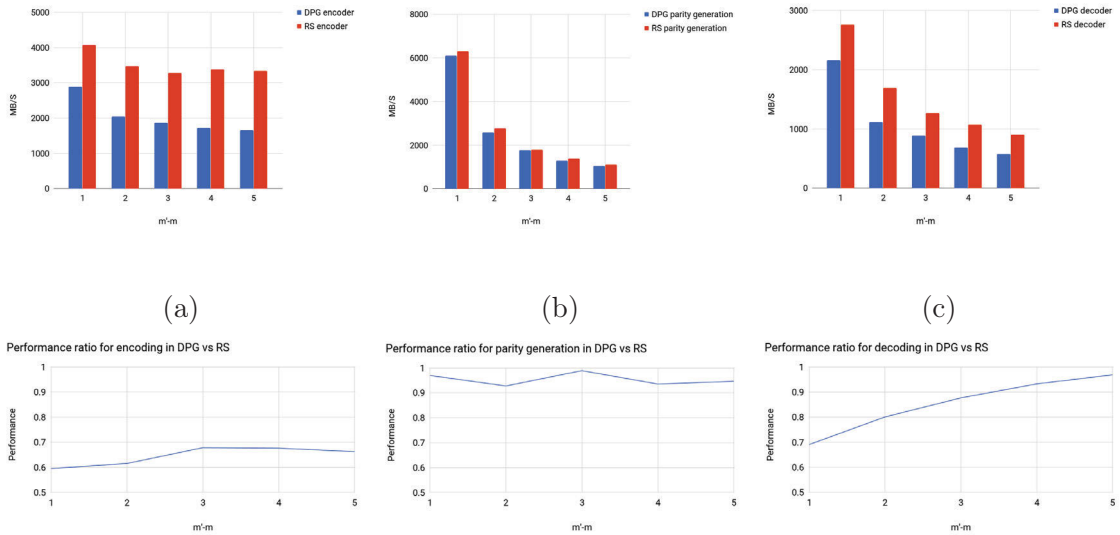
We tested the RS code and DPG for input files with different sizes. The result in shown in Figure 5.4. As it can be seen as the input file gets greater the encoding and decoding process takes longer and thus the performance reduces. However, from the ratio plots we see that the speed is almost constant and we have almost flat curves. We believe the only reason that the plots are not zero slop lines is that the time that



(a) (b) (c)

(d) (e) (f)

Figure 5.2: Comparing the performance of RS and DPG code for different number of initially-generated parities.



(a) (b) (c)

(d) (e) (f)

Figure 5.3: Comparing the performance of RS and DPG code for different number of delayed-generated parities.

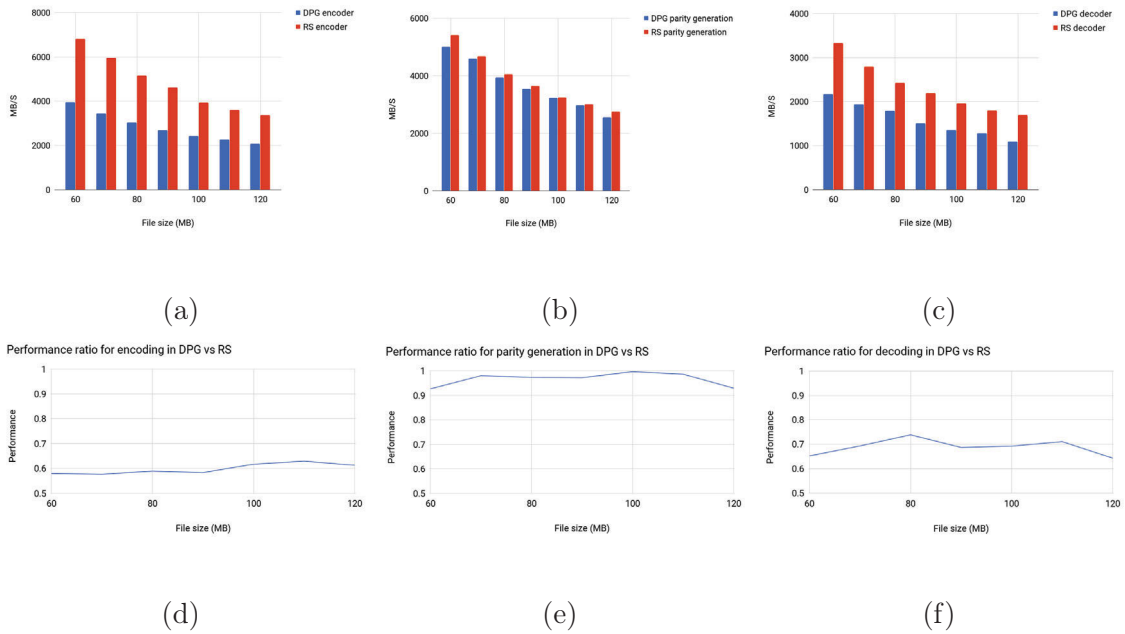


Figure 5.4: Comparing the performance of RS and DPG code for files with different sizes.

we get as the run-time is not %100 correct and other programs being run on the same system at the same time as our test could have affect our results.

As it is mentioned in the previous sections our proposed code is efficient and has low complexity in the delayed parity generation process. In all the ratio plots for delayed parity generation it is evident that DPG performs either similar to RS code or event better.

The results shown in the plots match the complexities of the codes for encoding and decoding mentioned in section 5.2.

CHAPTER 6

CONCLUSION AND DISCUSSION

We proposed delayed parity generation as a method to accelerate the write speed in erasure-coded data storage systems. By delaying the generation, transportation, and writing of some of the parities to time of light system load, this approach can improve the initial write (commit) speed, at the expense of a small loss of reliability during a short period of time. Through a connection to network coding, we identified the fundamental limits of such systems in terms of the minimum amount of disk-IO, and then proposed a novel explicit code construction. The proposed code construction has low computational complexity, thus does not increase the computation burden during write time or delayed parity generation. We also confirmed this experimentally. The connection to the adaptivity problem is also explored, where the proposed code can be applied. Our result shows that blindly adopting regenerating codes in either setting is unnecessary and wasteful, and the proposed code can accomplish the same functionalities with a conceptually simpler code, and with a much lower computational cost.

LIST OF REFERENCES

[1] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[2] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *ACM SIGOPS operating systems review*, volume 37, pages 29–43. ACM, 2003.

[3] Ranjita Bhagwan, Kiran Tati, Yuchung Cheng, Stefan Savage, and Geoffrey M Voelker. Total recall: System support for automated availability management. In *Nsdi*, volume 4, pages 25–25, 2004.

[4] Sean Rhea, Chris Wells, Patrick Eaton, Dennis Geels, Ben Zhao, Hakim Weatherspoon, and John Kubiatowicz. Maintenance-free global data storage. *IEEE internet computing*, 5(5):40–49, 2001.

[5] J Jeffrey Hanson. An introduction to the hadoop distributed file system. *IBM-United States. Np*, 1, 2011.

[6] Shu Lin and Daniel J Costello. *Error control coding*, volume 2. Prentice Hall Englewood Cliffs, 2004.

[7] Jianzhong Huang, Xianhai Liang, Xiao Qin, Ping Xie, and Changsheng Xie. Scale-rs: An efficient scaling scheme for rs-coded storage clusters. *IEEE Transactions on Parallel and Distributed Systems*, 26(6):1704–1717, 2015.

[8] Frank Dabek, Jinyang Li, Emil Sit, James Robertson, M Frans Kaashoek, and Robert Morris. Designing a dht for low latency and high throughput. In *NSDI*, volume 4, pages 85–98, 2004.

[9] Sean C Rhea, Patrick R Eaton, Dennis Geels, Hakim Weatherspoon, Ben Y Zhao, and John Kubiatowicz. Pond: The oceanstore prototype. In *FAST*, volume 3, pages 1–14, 2003.

[10] Hakim Weatherspoon, John Kubiatowicz, et al. Erasure coding vs. replication: A quantitative comparison. In *IPTPS*, volume 1, pages 328–338. Springer, 2002.

[11] Michael G Luby, Michael Mitzenmacher, Mohammad Amin Shokrollahi, and Daniel A Spielman. Improved low-density parity-check codes using irregular graphs. *IEEE Transactions on information Theory*, 47(2):585–598, 2001.

[12] Amin Shokrollahi. Raptor codes. *IEEE transactions on information theory*, 52(6):2551–2567, 2006.

[13] Michael Luby. Lt codes. In *Foundations of Computer Science, 2002. Proceedings. The 43rd Annual IEEE Symposium on*, pages 271–280. IEEE, 2002.

[14] Lihao Xu and Jehoshua Bruck. X-code: Mds array codes with optimal encoding. *IEEE Transactions on Information Theory*, 45(1):272–276, 1999.

[15] Mario Blaum, Jim Brady, Jehoshua Bruck, and Jai Menon. Evenodd: An efficient scheme for tolerating double disk failures in raid architectures. *IEEE Transactions on computers*, 44(2):192–202, 1995.

[16] James Lee Hafner. Weaver codes: Highly fault tolerant erasure codes for storage systems. In *FAST*, volume 5, pages 16–16, 2005.

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

[18] 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.

[19] David A Patterson, Garth Gibson, and Randy H Katz. *A case for redundant arrays of inexpensive disks (RAID)*, volume 17. ACM, 1988.

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

[21] HDFS Erasure Coding. https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HDFSErasureCoding.html. Accessed: 2017-07-15.

[22] Alexandros G Dimakis, P Brighten Godfrey, Yunnan Wu, Martin J Wainwright, and Kannan Ramchandran. Network coding for distributed storage systems. *IEEE Transactions on Information Theory*, 56(9):4539–4551, 2010.

[23] Viveck R Cadambe, Syed Ali Jafar, Hamed Maleki, Kannan Ramchandran, and Changho Suh. Asymptotic interference alignment for optimal repair of mds codes in distributed storage. *IEEE Transactions on Information Theory*, 59(5):2974–2987, 2013.

[24] Nihar B Shah, K Vinayak Rashmi, P Vijay Kumar, and Kannan Ramchandran. Distributed storage codes with repair-by-transfer and nonachievability of interior points on the storage-bandwidth tradeoff. *IEEE Transactions on Information Theory*, 58(3):1837–1852, 2012.

[25] Nihar B Shah, KV Rashmi, P Vijay Kumar, and Kannan Ramchandran. Interference alignment in regenerating codes for distributed storage: Necessity and code constructions. *IEEE Transactions on Information Theory*, 58(4):2134–2158, 2012.

[26] Korlakai Vinayak Rashmi, Nihar B Shah, and P Vijay Kumar. Optimal exact-regenerating codes for distributed storage at the msr and mbr points via a product-matrix construction. *IEEE Transactions on Information Theory*, 57(8):5227–5239, 2011.

[27] Min Ye and Alexander Barg. Explicit constructions of high-rate mds array codes with optimal repair bandwidth. *IEEE Transactions on Information Theory*, 63(4):2001–2014, 2017.

[28] Itzhak Tamo, Zhiying Wang, and Jehoshua Bruck. Access versus bandwidth in codes for storage. *IEEE Transactions on Information Theory*, 60(4):2028–2037, 2014.

[29] Y Hu. The mds scaling problem for cloud storage. In *Presentation at First Workshop on Network Coding and Data Storage*, 2011.

[30] Florence Jessie MacWilliams and Neil James Alexander Sloane. *The theory of error-correcting codes*. Elsevier, 1977.

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

[32] Kenneth W Shum and Yuchong Hu. Exact minimum-repair-bandwidth cooperative regenerating codes for distributed storage systems. In *Information Theory Proceedings (ISIT), 2011 IEEE International Symposium on*, pages 1442–1446. IEEE, 2011.

[33] Yuchong Hu, Yinlong Xu, Xiaozhao Wang, Cheng Zhan, and Pei Li. Cooperative recovery of distributed storage systems from multiple losses with network coding. *IEEE Journal on Selected Areas in Communications*, 28(2), 2010.

[34] Brijesh Kumar Rai, Vommi Dhoorjati, Lokesh Saini, and Amit K Jha. On adaptive distributed storage systems. In *Information Theory (ISIT), 2015 IEEE International Symposium on*, pages 1482–1486. IEEE, 2015.

[35] Huayu Zhang, Hui Li, Bing Zhu, Xin Yang, and Shuo-Yen Robert Li. Minimum storage regenerating codes for scalable distributed storage. *IEEE Access*, 5:7149–7155, 2017.

[36] Rudolf Ahlswede, Ning Cai, S-YR Li, and Raymond W Yeung. Network information flow. *IEEE Transactions on information theory*, 46(4):1204–1216, 2000.

[37] S-YR Li, Raymond W Yeung, and Ning Cai. Linear network coding. *IEEE transactions on information theory*, 49(2):371–381, 2003.

[38] R. E. Barlow and K. D. Heidtmann. Computing k-out-of-n system reliability. *IEEE Transactions on Reliability*, R-33(4):322–323, Oct 1984.

[39] J. E. Angus. On computing mtbf for a k-out-of-n:g repairable system. *IEEE Transactions on Reliability*, 37(3):312–313, Aug 1988.

[40] Peter M. Chen, Edward K. Lee, Garth A. Gibson, Randy H. Katz, and David A. Patterson. Raid: High-performance, reliable secondary storage. *ACM Comput. Surv.*, 26(2):145–185, June 1994.

[41] Jason K. Resch and Ilya Volvovski. Reliability models for highly fault-tolerant storage systems. *CoRR*, abs/1310.4702, 2013.

[42] Intel® streaming simd extensions technology, July 2017.

[43] J. S. Plank, E. L. Miller, and W. B. Houston. GF-Complete: A comprehensive open source library for Galois Field arithmetic. Technical Report UT-CS-13-703, University of Tennessee, January 2013.

[44] J. S. Plank. Jerasure: A library in C/C++ facilitating erasure coding for storage applications. Technical Report CS-07-603, University of Tennessee, September 2007.

APPENDIX

APPENDIX A

THE CODE

In this appendix, the code used for encoding, delayed parity generation and decoding is presented. In this implementation, m is the total number of parities, m_1 is the initial number of parities and $m_2 = m - m_1$. In other words, in the C code m_1, m and $m2$ are equivalent to m, m' and $m' - m$ in the previous chapters.

A.1 Encoder

```
#include <assert.h>
#include <sys/time.h>
#include <time.h>
#include <sys/time.h>
#include <sys/stat.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include "jerasure.h"
#include "reed_sol.h"
#include "timing.h"
#include <math.h>
#include <unistd.h>
#include <stdint.h>
#include <stdio.h>
#include "tools.h"

int main (int argc, char **argv) {
```

```
FILE *fp, *fp2;        // file pointers
char *block; // padding file
int size, newsize, original_size; // size of file and temp
↪  size

struct stat status; // finding file size

enum Coding_Technique tech;        // coding technique
↪  (parameter)
int k, m, m1, m2, w; // parameters k=number of systematic
↪  nodes, m1=number of initial parities, m2=number of delayed
↪  parities, m=m1+m2.
int buffersize;        // paramter
int i, j; // loop control variables
int blocksize; // size of k+m files
int total;
int extra;
double readins;

/* DPG Arguments */
char ***data; //// changed it to triple pointer instead of 2
char ***coding; //// the same as data
int *matrix;

/* Creation of file name variables */
char temp[5];
char *s1, *s2, *extension;
char *fname;
int md;
char *curdir;

/* Timing variables */
uint64_t prev_time, time_value, time_diff;
time_diff = 0;
```

```
matrix = NULL;

/* Error check Arguments*/
if (argc != 8) {
        fprintf(stderr,  "usage: inputfile k m1 m2 w
        ↪  buffersize dest_dir_path(m1: initial parities, m2:
        ↪  later added parities)\n");
        fprintf(stderr,  "\nBuffersize of 0 means the
        ↪  buffersize is chosen automatically.\n");
        exit(0);
}
/* Conversion of parameters and error checking */
if (sscanf(argv[2], "%d", &k) == 0 || k <= 0) {
        fprintf(stderr,  "Invalid value for k\n");
        exit(0);
}
if (sscanf(argv[3], "%d", &m1) == 0 || m1 < 0) {
        fprintf(stderr,  "Invalid value for m1\n");
        exit(0);
}
if (sscanf(argv[4], "%d", &m2) == 0 || m2 < 0) {
        fprintf(stderr,  "Invalid value for m2\n");
        exit(0);
}
if (sscanf(argv[5],"%d", &w) == 0 || w <= 0) {
        fprintf(stderr,  "Invalid value for w.\n");
        exit(0);
}
if (sscanf(argv[6],"%d", &buffersize) == 0 || buffersize < 0)
↪  {
        fprintf(stderr,  "Invalid value for buffersize.\n");
        exit(0);
}
```

```
m = m1+m2; //here I consider m to be the total number of
↪  parities(overall)
if (argv[1][0] != '-') {
    /* Open file and error check */
    fp = fopen(argv[1], "rb");
    if (fp == NULL)
    {
        fprintf(stderr,  "Unable to open file.\n");
        exit(0);
    }

    /* Get current working directory for construction of file
    ↪  names */
    curdir = (char*)malloc(sizeof(char)*1000);
    sprintf(curdir, argv[7]);
    char *name;
    name = (char *)malloc(sizeof(char)*(strlen(argv[7]+20)));
    sprintf(name, "%s/Coding", curdir);

    /* Create Coding directory */
    i = mkdir(name, S_IRWXU);
    if (i == -1 && errno != EEXIST)
    {
        fprintf(stderr, "Unable to create Coding directory.\n");
        exit(0);
    }

    fclose(fp);
    /* Determine original size of file */
    stat(argv[1], &status);
    size = status.st_size;
    original_size = size;
    fp = fopen(argv[1], "rb");
}
```

```
else
{
    if (sscanf(argv[1]+1, "%d", &size) != 1 || size <= 0)
    {
        fprintf(stderr, "Files starting with '-' should be sizes
         ↪  for randomly created input\n");
        exit(1);
    }
    fp = NULL;
    MOA_Seed(time(0));
}
if(buffersize == 0)
{
    size = pad_file(size, m, k*w*sizeof(long));
    size = size/m;
    buffersize = size;
    blocksize = size/(k);
    readins = 1;
}
else
{
            size = pad_file(size, m, k*w*sizeof(long));
            buffersize = size;
    readins = ceil((double)(size /(double)(buffersize)));
            buffersize = buffersize/m;
            size = buffersize;
    blocksize = buffersize/k;
```

```
                //the commented out part is how I was pading based on
                ↪  the buffer size that user provides. But for
                ↪  checking the performance sice Dr. Plank in
                ↪  jerasure was increasing the buffer size such that
                ↪  readins was always 1 (at least for the files that
                ↪  I have tested the code with), I changed my padding
                ↪  approach too. So, that every thing is done in one
                ↪  step(i.g. readins = 1)

        /*buffersize = pad_file(buffersize, m, k*w*sizeof(long));
        readins = ceil((double)(original_size
↪  /(double)(buffersize)));
                buffersize = buffersize/m;
        size = buffersize;//size is the size of the data for one of
↪  the instaces within a specific readins.
        blocksize = buffersize/k;*/

    }
        printf(", %d,", buffersize);
    block = (char *)malloc(sizeof(char)*buffersize);
        /* Setting of coding technique and error checking */
        tech = Reed_Sol_Van;

        /* Break inputfile name into the filename and extension */
        s1 = (char*)malloc(sizeof(char)*(strlen(argv[1])+20));
        s2 = strrchr(argv[1], '/');
        if (s2 != NULL) {
                s2++;
                strcpy(s1, s2);
        }
        else {
                strcpy(s1, argv[1]);
        }
        s2 = strchr(s1, '.');
```

```
    if (s2 != NULL) {
      extension = strdup(s2);
      *s2 = '\0';
    } else {
      extension = strdup("");
}

    /* Allocate for full file name */
    fname =
    ↪  (char*)malloc(sizeof(char)*(strlen(argv[1])+strlen(curdir)+20));
    sprintf(temp, "%d", k);
    md = strlen(temp);

    /* Allocate data and coding */
data = mem_aloc_3d(m, k, blocksize);
    coding = mem_aloc_3d(m, m, blocksize);

    /* Create coding matrix or bitmatrix and schedule */
    prev_time = get_posix_clock_time();

    matrix = reed_sol_vandermonde_coding_matrix(k, m, w);

    time_value = get_posix_clock_time();
    time_diff += time_value - prev_time;

int total_byte_read, n;
total_byte_read = 0;
n = 1;
while(n <= readins)
{
    /* Read in data until finished */
    int r;
    for(r = 0; r < m; r++)
    {
```

```
total = 0;
    /* Check if padding is needed, if so, add appropriate
    ↪  number of zeros. Each time a data read from the
    ↪  file is saved in block and is divided between k
    ↪  nodes(data[c][0], data[c][1],...,data[c][k-1]. C
    ↪  is the column index).*/
    if (total < size && total + buffersize <= size &&
    ↪  total_byte_read + buffersize <= original_size) {
        total += jfread(block, sizeof(char), buffersize,
        ↪  fp);
    }
    else if (total < size && total_byte_read + buffersize
    ↪  > original_size) {
        extra = jfread(block, sizeof(char), buffersize,
        ↪  fp);
        for (i = extra; i < buffersize; i++) {
            block[i] = '0';
        }
    }
    else if (total == size) {
        for (i = 0; i < buffersize; i++) {
            block[i] = '0';
        }
    }
    for (i = 0; i < k; i++)
    {
        memcpy(data[r][i], block+(i*blocksize),
        ↪  blocksize);
        total_byte_read += blocksize;
    }

                    prev_time = get_posix_clock_time();
    /* Encode */
```

```
        if(r < m1) //because for the first m1 columns all
        ↪  parities need to be generated
        {

            jerasure_matrix_encode(k, m, w, matrix, data[r],
            ↪  coding[r], blocksize);
        }
        else // //because for the last m-m1 columns only the
        ↪  first m1 parities need to be generated
        {
            jerasure_matrix_encode(k, m1, w, matrix, data[r],
            ↪  coding[r], blocksize);
        }
                        time_value = get_posix_clock_time();
                        time_diff += time_value - prev_time;
}
        //Here we have all parities that we need, and we XOR
        ↪  the desired ones
        prev_time = get_posix_clock_time();
for(i = 0; i < m1; i++)
{
    for(j = m1; j < m; j++ )
    {
        galois_region_xor(coding[i][j], coding[j][i],
        ↪  blocksize); //galois_region_xor(char * src, char *
        ↪  dest, int nbyte) result is saved in dest
    }
}

        time_value = get_posix_clock_time();
        time_diff += time_value - prev_time;
        // Writing the results in the corresponding file
for(i = 1; i <= k; i++){
```

```
        sprintf(fname, "%s/Coding/%s_systematic_node%0*d%s",
        ↪  curdir, s1, md, i, extension);
        if(n == 1)
            fp2 = fopen(fname, "wb");
        else
            fp2 = fopen(fname, "ab");
        for        (j = 0; j < m; j++)
            fwrite(data[j][i-1], sizeof(char), blocksize, fp2);
        fclose(fp2);
    }
    for(i = 1; i <= m1; i++){
        sprintf(fname, "%s/Coding/%s_parity_node%0*d%s", curdir,
        ↪  s1, md, i, extension);
        if(n == 1)
            fp2 = fopen(fname, "wb");
        else
            fp2 = fopen(fname, "ab");
        for        (j = 0; j < m; j++)
            fwrite(coding[j][i-1], sizeof(char), blocksize, fp2);
        fclose(fp2);
    }
    n++;
}
/* Create metadata file */
    if (fp != NULL) {
    sprintf(fname, "%s/Coding/%s_meta.txt", curdir, s1);
    fp2 = fopen(fname, "wb");
    fprintf(fp2, "%s\n", argv[1]);
    fprintf(fp2, "%d\n", size);
    fprintf(fp2, "%d %d %d %d %d\n", k, m1, m2, w, buffersize);
    fprintf(fp2, "%s\n", "Reed_Sol_Van");
    fprintf(fp2, "%d\n", tech);
    fprintf(fp2, "%d\n", (int)readins);
    fprintf(fp2, "%d\n", original_size);
```

```
        fclose(fp2);
    }
    fclose(fp);
        /* Free allocated memory */
        free(s1);
        free(fname);
        free(block);
        free(curdir);
    free(extension);
    free(matrix);

    free_mem_3d(coding, m, m, blocksize);
    free_mem_3d(data, m, k, blocksize);

        /* Calculate time in second and print */
        printf(" %f, ", (double)(time_diff/1000000.0));
        return 0;

}
```

A.2 Delayed parity generation

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <assert.h>
#include <unistd.h>
#include <sys/time.h>
#include <sys/stat.h>
#include <signal.h>
#include <unistd.h>
```

```
#include "jerasure.h"
#include "reed_sol.h"
#include "timing.h"
#include "tools.h"


int main (int argc, char **argv) {
        FILE *fp;                                   // File pointer

        /* DPG arguments */
        char ***data;// data and coding are used to store the data
        ↪  read in them. data size = m2*k*blocksize and coding size =
        ↪  m2*m1*blocksize
        char ***coding;
    char ***new_parities; // for new created parities with size
    ↪  m2*m*blocksize

        int *matrix;

        /* Parameters */
        int k, m, m1, m2, w, buffersize;
        int tech;
        char *c_tech;

        int i, j, z, r;                                 // loop control
        ↪  variable, s
        int blocksize = 0;                          // size of
        ↪  individual files
        int origsize,
        ↪  origin_size_before_padding;                          //
        ↪  origsize = size of file for each instane =
        ↪  padded_file_size/m.
        int total, total_decoded;                                 //
        ↪  used to write data, not padding to file
```

```
    struct stat status;                 // used to find size of
    ↪  individual files
    int numerased;                          // number of erased
    ↪  files
int n;
    int readins = 0;

    /* Used to recreate file names */
    char *temp;
    char *cs1, *cs2, *extension;
    char *fname;
    int md;
    char *curdir;

    /* Used to time decoding */
    uint64_t prev_time, time_value, time_diff;
    time_diff = 0;

    matrix = NULL;

    /* Error checking parameters */
    if (argc != 3) {
            fprintf(stderr, "usage: inputfile\n");
            exit(0);
    }
    curdir = (char *)malloc(sizeof(char)*1000);
    //assert(curdir == getcwd(curdir, 1000));
    sprintf(curdir, argv[2]);

    /* Begin recreation of file names */
    cs1 = (char*)malloc(sizeof(char)*strlen(argv[1]));
    cs2 = strrchr(argv[1], '/');
    if (cs2 != NULL) {
            cs2++;
```

```
        strcpy(cs1, cs2);
}
else {
        strcpy(cs1, argv[1]);
}
cs2 = strchr(cs1, '.');
if (cs2 != NULL) {
        extension = strdup(cs2);
        *cs2 = '\0';
} else {
   extension = strdup("");
}
fname = (char
↪  *)malloc(sizeof(char*)*(100+strlen(argv[1])+20));

/* Read in parameters from metadata file */
sprintf(fname, "%s/Coding/%s_meta.txt", curdir, cs1);

fp = fopen(fname, "rb");
if (fp == NULL) {
  fprintf(stderr, "Error: no metadata file %s\n", fname);
  exit(1);
}
temp = (char *)malloc(sizeof(char)*(strlen(argv[1])+20));
if (fscanf(fp, "%s", temp) != 1) {
        fprintf(stderr, "Metadata file - bad format\n");
        exit(0);
}

if (fscanf(fp, "%d", &origsize) != 1) {
        fprintf(stderr, "Original size is not valid\n");
        exit(0);
}
```

```
    if (fscanf(fp, "%d %d %d %d %d", &k, &m1, &m2, &w ,
    ↪  &buffersize) != 5) {
            fprintf(stderr, "Parameters are not correct\n");
            exit(0);
    }
    c_tech = (char *)malloc(sizeof(char)*(strlen(argv[1])+20));
    if (fscanf(fp, "%s", c_tech) != 1) {
            fprintf(stderr, "Metadata file - bad format\n");
            exit(0);
    }
    if (fscanf(fp, "%d", &tech) != 1) {
            fprintf(stderr, "Metadata file - bad format\n");
            exit(0);
    }
    method = tech;
    if (fscanf(fp, "%d", &readins) != 1) {
            fprintf(stderr, "Metadata file - bad format\n");
            exit(0);
    }
    if (fscanf(fp, "%d", &origin_size_before_padding) != 1) {
            fprintf(stderr, "Original size before padding is not
            ↪  valid\n");
            exit(0);
    }
    //setting paramaters and allocating memory
    fclose(fp);
m = m1 + m2;
blocksize = buffersize/k;
data = mem_aloc_3d(m2, k, blocksize);
coding = mem_aloc_3d(m2, m1, blocksize);
    sprintf(temp, "%d", k);
    md = strlen(temp);
    prev_time = get_posix_clock_time();
```

```
            matrix = reed_sol_vandermonde_coding_matrix(k, m, w);
            ↪  //generating the coding matrix
    time_value = get_posix_clock_time();
    time_diff += time_value - prev_time;

n = 1;
while(n<=readins)
{
            //Reading the required data
    //I'm gonna read from the systematic data here
    for(i = 1; i <=k; i++)
    {
        sprintf(fname, "%s/Coding/%s_systematic_node%0*d%s",
        ↪  curdir, cs1, md, i , extension);
        fp = fopen(fname, "rb");
         if (fp == NULL) {
            printf("\nCould not find this file\n");
            exit(-1);
        }
        else {

                for(j = 0; j < m2; j++){
                    data[j][i-1] = (char
                    ↪  *)malloc(sizeof(char)*blocksize);
                    fseek(fp, ((n-1)*(m1+m2)*blocksize) +
                    ↪  blocksize*(m1+j), SEEK_SET);
                    assert(blocksize == fread(data[j][i-1],
                    ↪  sizeof(char), blocksize, fp));
                }

            fclose(fp);
        }

    }
```

```
//I'm gonna read from the parities here
for(i = 1; i <=m1; i++)
{
    sprintf(fname, "%s/Coding/%s_parity_node%0*d%s", curdir,
    ↪  cs1, md, i , extension);
    fp = fopen(fname, "rb");
    if (fp == NULL) {
        printf("\nCould not find this file\n");
        exit(-1);
    }
    else {
            for(j = 0; j < m2; j++){
                coding[j][i-1] = (char
                ↪  *)malloc(sizeof(char)*blocksize);
                fseek(fp, ((n-1)*(m1+m2)*blocksize) +
                ↪  blocksize*(m1+j), SEEK_SET);
                assert(blocksize == fread(coding[j][i-1],
                ↪  sizeof(char), blocksize, fp));
            }
             fclose(fp);
    }
}
/* All the reading from files is done here. It is important to
↪  note that although I am reading the (m1)th column (because
↪  the index is from 0) but it is stored in the 0 index of
↪  codeing and data and so on*/

new_parities = mem_aloc_3d(m, m2, blocksize);
        prev_time = get_posix_clock_time();
generating_parities(k, w, m2, m1, m,  matrix, data, coding,
↪  new_parities, blocksize);
        time_value = get_posix_clock_time();
        time_diff += time_value - prev_time;
```

```
    for( i = 1; i <= m2; i++)//parity
    {
        sprintf(fname,"%s/Coding/%s_parity_node%0*d%s", curdir,
        ↪  cs1, md, m1+i, extension);
        if(n == 1)
            fp = fopen(fname, "wb");
        else
            fp = fopen(fname, "ab");
        for(j = 0; j < m; j++)//columns
        {
            fwrite(new_parities[j][i-1], sizeof(char), blocksize,
            ↪  fp);
        }
        fclose(fp);
    }
    n++;
}
/* Update metadata file. Here instead of updating the meta data
↪  file, I'm going to overwrite it. Because the decoder will read
↪  this file and needs to know that the new parities have been
↪  created. Meaning m1 = m1+m2 and m2 = 0 */
m1 = m1 + m2;
m2 = 0;
if (fp != NULL) {
    sprintf(fname, "%s/Coding/%s_meta.txt", curdir, cs1);
    fp = fopen(fname, "wb");
    fprintf(fp, "%s\n", argv[1]);
    fprintf(fp, "%d\n", origsize);
    fprintf(fp, "%d %d %d %d %d\n", k, m1, m2, w, buffersize);
    fprintf(fp, "%s\n", c_tech);
    fprintf(fp, "%d\n", tech);
    fprintf(fp, "%d\n", readins);
    fprintf(fp, "%d\n", origin_size_before_padding);
    fclose(fp);
```

```
    }

    free(cs1);
    free(extension);
    free(fname);
    free(temp);
    free(matrix);
    free(curdir);
    free(c_tech);

    free_mem_3d(data, m2, k, blocksize);
    free_mem_3d(coding, m2, m1, blocksize);
    free_mem_3d(new_parities, m, m2, blocksize);

        printf(" %f, ",((double)time_diff)/1000000.0);

    return 0;
}
```

A.3 Decoder

Here are the two main functions used in the decoder program.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <assert.h>
#include <sys/time.h>
#include <sys/stat.h>
#include "jerasure.h"
#include "reed_sol.h"
#include "timing.h"
```

```c
#include <unistd.h>
#include <stdint.h>
#include <stdio.h>

#include "tools.h"

int main (int argc, char **argv) {
        FILE *fp;                                   // File pointer

        char ***data;// I changed data and coding to 3 dimantional
        ↪  objects
        char ***coding;
    char ***coding_copy;
        int **erasures;//I changed this and erased to 2 dimention as
        ↪  well so I could keep them for all instances
        int **erased;
        int *matrix;

    int **parities_metadata; //I use this to determine the index of
    ↪  all parities that I have within m1 parities
    int **coding_table_status; // I use this table to know the status
    ↪  of the codes. 0 for the erased ones and 1 for the existing
    ↪  ones.

        /* Parameters */
        int k, m, m1, m2, w, buffersize;
        int tech;
        char *c_tech;

        int i, j, z;                                  // loop control
        ↪  variable, s
        int blocksize = 0;                          // size of
        ↪  individual files
```

```
    int origsize,
    ↪ origin_size_before_padding;                        // size
    ↪ of file before padding, origsize is the size of each of
    ↪ the columns
    int total_decoded;                                  // used to
    ↪ write data, excluding the padding part to file
    struct stat status;                // used to find size of
    ↪ individual files
    int numerased;                        // number of erased
    ↪ files
int byte_num;
byte_num = 0;// is used to count the number of bytes from the
↪ decoded files
int readins, n;

    /* Used to recreate file names */
    char *temp;
    char *cs1, *cs2, *extension;
    char *fname;
    int md;
    char *curdir;

    /* Used to time decoding */
    uint64_t prev_time, time_value, time_diff;
    time_diff = 0;

    matrix = NULL;

/* Error checking parameters */
    if (argc != 3) {
            fprintf(stderr, "usage: inputfile\n");
            exit(0);
    }
    curdir = (char *)malloc(sizeof(char)*1000);
```

```
sprintf(curdir, argv[2]);

/* Begin recreation of file names */
cs1 = (char*)malloc(sizeof(char)*strlen(argv[1]));
cs2 = strrchr(argv[1], '/');
if (cs2 != NULL) {
        cs2++;
        strcpy(cs1, cs2);
}
else {
        strcpy(cs1, argv[1]);
}
cs2 = strchr(cs1, '.');
if (cs2 != NULL) {
        extension = strdup(cs2);
        *cs2 = '\0';
} else {
   extension = strdup("");
}
fname = (char
↪  *)malloc(sizeof(char*)*(100+strlen(argv[1])+20));

/* Read in parameters from metadata file */
sprintf(fname, "%s/Coding/%s_meta.txt", curdir, cs1);

fp = fopen(fname, "rb");
if (fp == NULL) {
  fprintf(stderr, "Error: no metadata file %s\n", fname);
  exit(1);
}
temp = (char *)malloc(sizeof(char)*(strlen(argv[1])+20));
if (fscanf(fp, "%s", temp) != 1) {
        fprintf(stderr, "Metadata file - bad format\n");
        exit(0);
```

```
    }

    if (fscanf(fp, "%d", &origsize) != 1) {
            fprintf(stderr, "Original size is not valid\n");
            exit(0);
    }
    if (fscanf(fp, "%d %d %d %d %d", &k, &m1, &m2, &w,
     ↪  &buffersize)!= 5) {
            fprintf(stderr, "Parameters are not correct\n");
            exit(0);
    }
    c_tech = (char *)malloc(sizeof(char)*(strlen(argv[1])+20));
    if (fscanf(fp, "%s", c_tech) != 1) {
            fprintf(stderr, "Metadata file - bad format\n");
            exit(0);
    }
    if (fscanf(fp, "%d", &tech) != 1) {
            fprintf(stderr, "Metadata file - bad format\n");
            exit(0);
    }
    method = tech;
    if (fscanf(fp, "%d", &readins) != 1) {
            fprintf(stderr, "Metadata file - bad format\n");
            exit(0);
    }
    if (fscanf(fp, "%d", &origin_size_before_padding) != 1) {
            fprintf(stderr, "Original size is not valid\n");
            exit(0);
    }
    fclose(fp);
m = m1 + m2;
    /* Allocate memory */
    erased = (int **)malloc(sizeof(int*)*m);
for (i = 0; i < m; i++)
```

```
    erased[i] = (int *)malloc(sizeof(int)*(k+m));

    erasures = (int **)malloc(sizeof(int*)*m);
for (i = 0; i < m; i++)
    erasures[i] = (int *)malloc(sizeof(int)*(k+m));

blocksize = buffersize/k;
data = mem_aloc_3d(m, k, blocksize);
coding = mem_aloc_3d(m, m, blocksize);

    sprintf(temp, "%d", k);
    md = strlen(temp);

    prev_time = get_posix_clock_time();
            matrix = reed_sol_vandermonde_coding_matrix(k, m,
             ↪  w);//Here I am only using vandermonde matrix
    time_value = get_posix_clock_time();
    time_diff += time_value - prev_time;

data = mem_aloc_3d(m, k, blocksize);
    coding = mem_aloc_3d(m, m, blocksize);

    /* Begin decoding process */
total_decoded = 0;
n = 1;
while(n <= readins)
{
    for (i = 0; i < m; i++)
        for (j = 0; j < k+m; j++)
                    erased[i][j] = 0;

    numerased = -1;
    int r;
    for (i = 1; i <= k; i++) {
```

```
        sprintf(fname, "%s/Coding/%s_systematic_node%0*d%s",
        ↪  curdir, cs1, md, i , extension);
        fp = fopen(fname, "rb");
        if (fp == NULL) {
            numerased++;
            for(j = 0; j < m; j++)
            {
                erased[j][i-1] = 1;
                erasures[j][numerased] = i-1;
            }
        }
        else {
                for(j = 0; j < m; j++)
                {
                    fseek(fp, ((n-1)*m)*blocksize+j*blocksize,
                    ↪  SEEK_SET);
                    assert(blocksize == fread(data[j][i-1],
                    ↪  sizeof(char), blocksize, fp));
                }
            fclose(fp);
        }
    }
    for (i = 1; i <= m; i++) {
        sprintf(fname, "%s/Coding/%s_parity_node%0*d%s", curdir,
        ↪  cs1, md, i , extension);
        fp = fopen(fname, "rb");
        if (fp == NULL) {
            numerased++;
            for(j = 0; j < m; j++)
            {
                erased[j][k+i-1] = 1;
                erasures[j][numerased] = k+i-1;
            }
        }
```

```
        else {
                for(j = 0; j < m; j++)
                {
                    fseek(fp,((n-1)*m)*blocksize+j*blocksize,
                    ↪  SEEK_SET);
                                                int temp;
                                                temp =
                                                ↪  fread(coding[j][i-1],
                                                ↪  sizeof(char),
                                                ↪  blocksize, fp);
                                                assert(blocksize ==
                                                ↪  temp);
                }
            fclose(fp);
        }
    }
    for(i = 0; i < m; i++)
        erasures[i][numerased+1] = -1;

            prev_time = get_posix_clock_time();
            //Decoding starts here
            //the first m1 columns already have what the need to
            ↪  decode and reconstruct. Using
            ↪  reconstraction_using_single_symbols I first
            ↪  reconstruct the lost part of the systematic nodes
            ↪  for column i, then I use
            ↪  regenerating_requierd_single_nodes to regenerate
            ↪  the lost parities for column i. Using the resulted
            ↪  parities, I xor them with the symbols that are in
            ↪  the symetric location of them.
            for (i = 0; i < m1; i++)
            {
        reconstraction_using_single_symbols(i, k, m, w, matrix,
        ↪  coding[i], data[i], erasures[i], blocksize);
```

```
    regenerating_requierd_single_nodes(k, w, m, matrix,
     ↪  data[i], coding[i], blocksize, erased[i]);
    for(j = m1; j < m; j++ )
    {
        galois_region_xor(coding[i][j], coding[j][i],
         ↪  blocksize); //galois_region_xor(char * src, char *
         ↪  dest, int nbyte) result is saved in dest
    }
        }
        //At this point the last m2 columns have what the need
         ↪  to reconstruct their systematic portion
        for (i = m1; i < m; i++)
        {
    reconstraction_using_single_symbols(i, k, m, w, matrix,
     ↪  coding[i], data[i], erasures[i], blocksize);
        }

        time_value = get_posix_clock_time();
        time_diff += time_value - prev_time; // In our timing
         ↪  we do not include I/O. It is only computation
         ↪  time.
/*Here I'm going to merge all decoded files*/
FILE *fp2;
if(n == 1)
{
    sprintf(fname, "%s/Coding/%sdecoded%s", curdir, cs1 ,
     ↪  extension);
    fp = fopen(fname, "wb");
}
else
{
    sprintf(fname, "%s/Coding/%sdecoded%s", curdir, cs1 ,
     ↪  extension);
    fp = fopen(fname, "ab");
```

```
    }
    for(i = 0; i < m; i++)
    {
        for(j = 0; j < k; j++)
        {
            if(byte_num + blocksize <= origin_size_before_padding)
            {
                fwrite(data[i][j], sizeof(char), blocksize, fp);
                byte_num += blocksize;
            }
            else
            {
                fwrite(data[i][j], sizeof(char),
                ↪  origin_size_before_padding - byte_num, fp);
                byte_num += (origin_size_before_padding-byte_num);
            }
        }
    }
    fclose(fp);
    n++;
}
    /* Free allocated memory */
    free(cs1);
    free(extension);
free(temp);
free(curdir);
    free(fname);
free(matrix);
free(c_tech);

free_mem_2d(erasures, m);
free_mem_2d(erased, m);

free_mem_3d(coding, m, m, blocksize);
```

```
    free_mem_3d(data, m, k, blocksize);
        //time in secodes
        printf(" %f", (double)(time_diff/1000000.0));

        return 0;
}
```

A.4 Functions and headers

Here are the functions that are used in the encoder, delayed parity generations and decoder.

A.4.1 Functions

```
#include <assert.h>
#include <time.h>
#include <sys/time.h>
#include <sys/stat.h>
#include <unistd.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <signal.h>
#include <gf_rand.h>
#include <unistd.h>
#include "jerasure.h"
#include "reed_sol.h"
#include "cauchy.h"
#include "liberation.h"
#include "timing.h"
#include "math.h"x
```

```c
#include <unistd.h>
#include <stdint.h>

#include "tools.h"

uint64_t get_posix_clock_time ()
{
    struct timespec ts;

    if (clock_gettime (CLOCK_MONOTONIC, &ts) == 0)
        return (uint64_t) (ts.tv_sec * 1000000 + ts.tv_nsec / 1000);
    else
        return 0;
}

int jfread(void *ptr, int size, int nmembers, FILE *stream)
{
  if (stream != NULL) return fread(ptr, size, nmembers, stream);

  MOA_Fill_Random_Region(ptr, size);
  return size;
}

void free_mem_3d(char *** place, int row, int col, int depth)
{
    int i , j;
    for( i = 0; i < row; i++)
    {
        for(j = 0; j < col; j++)
            free(place[i][j]);
        free(place[i]);
    }
    free(place);
}
```

```c
void free_mem_2d(int **place, int row)
{
    int i;
    for(i = 0; i < row; i++)
        free(place[i]);
    free(place);
}

char *** mem_aloc_3d(int row, int col, int depth)
{
    int i, j;
    char ***name = (char ***)malloc(sizeof(char**)*row);
    for (i = 0; i < row; i++)
    {
        name[i] = (char **)malloc(sizeof(char *)*col);
        for(j = 0; j < col ; j++)
        {
            name[i][j] = (char *)malloc(sizeof(char)*depth);
            if (name[i][j] == NULL)
            {
                perror("malloc"); exit(1);
            }
        }
    }
    return name;
}

int pad_file( int size, int m, int k )
{
    int padded_size;
    padded_size = size;
    /*padding the file so that it is devisible by k*m*/
    char temp = 0;
```

```
    if(padded_size%(m*k)!= 0)
    {
        while(padded_size %(m*k) != 0)//It was m first then I changed
        ↪  it
        {
            padded_size += 1;
        }
    }
    return padded_size;
}



//Decoder_functions:

double reconstraction_using_single_symbols(int instance, int k, int m,
↪  int w, int * matrix, char ** parity, char **data, int * erasures,
↪  int blocksize){
    int i, j;
    if(erasures[0] == 0)
        i = jerasure_matrix_decode(k, m, w, matrix, 0, erasures, data,
        ↪  parity, blocksize); /*Although we have m1 parities we need
        ↪  to work with m because of the decoding matrix that is
        ↪  going to be created*/
    else
        i = jerasure_matrix_decode(k, m, w, matrix, 1, erasures, data,
        ↪  parity, blocksize); /*Although we have m1 parities we need
        ↪  to work with m because of the decoding matrix that is
        ↪  going to be created*/
    if (i < 0)
    {
        printf("Can not decode. Maybe the number of survivor nodes are
        ↪  less than k nodes!");
        exit(0);
```

```
    }
    return 0;

}

void regenerating_requierd_single_nodes(int k, int w, int m, int *
↪   matrix, char **data, char ** parity, int blocksize, int
↪   *erased)//This 'parity' is not the real ones, it's a copy of it.
{
    int i, j;
    char **coding_temp;//could be one dimention but because
    ↪   jerasure_matrix_encode gets a char ** as an input so I make it
    ↪   char **.
    coding_temp = (char **)malloc(sizeof(char *)*1);
    coding_temp[0] = (char *)malloc(sizeof(char)*blocksize);
    for (i = 0; i < m; i++) //m shows the index of the parities
    {
        int * sub_matrix;
        sub_matrix = (int *)malloc(sizeof(int)*k);
        if (erased[k+i] == 1)
        {
            for(j = 0; j< k; j++)
                sub_matrix[j] = matrix[i*k+j];
            //"1" because one parity is generated at a time
            jerasure_matrix_encode(k, 1, w, sub_matrix, data,
            ↪   coding_temp, blocksize);
            //for(j = 0; j < blocksize; j++)
              //  parity[i][j] = coding_temp[0][j];
                        memcpy(parity[i], coding_temp[0], blocksize);
            erased[i] = 0;
            free(sub_matrix);
        }
        else
            free(sub_matrix);
```

```
    }
    free(coding_temp[0]);
    free(coding_temp);
}

//Parity_generator functions:

void generating_parities(int k, int w, int m2, int m1, int m, int *
↪   matrix, char ***data, char ***coding, char ***new_parities, int
↪   blocksize)
/*I am going to generate all parities for downloaded instaces that its
↪   data is in data*/
{
    int i, j, r;
    int * sub_matrix;
    char **coding_temp;
    coding_temp = (char **)malloc(sizeof(char *)*1);
    coding_temp[0] = (char *)malloc(sizeof(char)*blocksize);
    sub_matrix = (int *)malloc(sizeof(int)*k);

    for(j = 0; j < m; j++)//parity
    {
        for(r = 0; r< k; r++)
            sub_matrix[r] = matrix[j*k+r];
        for(i = 0; i < m2; i++) //column
        {
            if (j < m1)
            {
                jerasure_matrix_encode(k, 1, w, sub_matrix, data[i],
                ↪   coding_temp, blocksize);
                galois_region_xor(coding[i][j], coding_temp[0],
                ↪   blocksize);
                memcpy(new_parities[j][i], coding_temp[0], blocksize);
            }
```

```
            else
            {
                jerasure_matrix_encode(k, 1, w, sub_matrix, data[i],
                ↪  coding_temp, blocksize);
                memcpy(new_parities[i+m1][j-m1], coding_temp[0],
                ↪  blocksize);
            }
        }
    }
    free(sub_matrix);
    free(coding_temp[0]);
    free(coding_temp);
}
```

A.4.2 Headers

```
#ifndef TOOLS_H
#define TOOLS_H

enum Coding_Technique {Reed_Sol_Van, Reed_Sol_R6_Op, Cauchy_Orig,
↪  Cauchy_Good, Liberation, Blaum_Roth, Liber8tion, RDP, EVENODD,
↪  No_Coding};
enum Coding_Technique method;

//Encoder_functions
uint64_t get_posix_clock_time ();
int jfread(void *ptr, int size, int nmembers, FILE *stream);
void free_mem_3d(char *** place, int row, int col, int depth);
void free_mem_2d(int **place, int row);
char *** mem_aloc_3d(int row, int col, int depth);
int pad_file( int size, int m, int k );

//Decoder_functions
```

```
double reconstraction_using_single_symbols(int instance, int k, int m,
↪  int w, int * matrix, char ** parity, char **data, int * erasures,
↪  int blocksize);
void regenerating_requierd_single_nodes(int k, int w, int m, int *
↪  matrix, char **data, char ** parity, int blocksize, int *erased);

//Parity_generator functions
void generating_parities(int k, int w, int m2, int m1, int m, int *
↪  matrix, char ***data, char ***coding, char ***new_parities, int
↪  blocksize);

#endif
```

A.5 Dependencies

In order to run this program you need to have:

- GF-complete (http://lab.jerasure.org/jerasure/gf-complete)
- Jerasure (http://lab.jerasure.org/jerasure/jerasure)

VITA

Sara Mousavi was born in Kermanshah, Iran. She received her BS in software engineering from Razi university in Kermanshah, Iran in 2012. Sara attended the University of Tennessee, Knoxville in Fall 2016 working on erasure codes and data storage systems under Dr. Chao Tian, pursuing an MS in computer science. Following graduation, Sara will continue her PhD in software engineering under the supervision of Dr. Audris Mockus.