1-1-2011

Hyfs: design and implementation of a reliable file system

Jianqiang Luo
Wayne State University,

HYFS: DESIGN AND IMPLEMENTATION OF A RELIABLE FILE SYSTEM

by

JIANQIANG LUO

DISSERTATION

Submitted to the Graduate School

of Wayne State University,

Detroit, Michigan

in partial fulfillment of the requirements

for the degree of

DOCTOR OF PHILOSOPHY

2011

MAJOR: COMPUTER SCIENCE

Approved by:

_____

Advisor Date

_____

_____

_____

_____

DEDICATION

Dedicated to my family

ACKNOWLEDGMENTS

I am deeply grateful to my advisor, Dr. Lihao Xu. Over the past five years, he let me know what research is about and how to do it. He guided me through such a wonderful experience. Besides that, he also taught me to have a deep understanding of a problem and then have a good solution for it. Every time when we have a discussion, I am always inspired by his insightful views. I believe all these things I learned from him would be of great help for my future career. Of course, it is really fun to work with him.

I should express my sincere thanks for Mochan Shrestha, my lab mate. He created a friendly lab environment, so that we can freely discuss immature ideas. His mathematics background also impressed me. From him, I learned how mathematics is useful for research.

I am indebted to Dr. Cheng Huang and Dr. Philip Shilane, who offered me opportunities to work for Microsoft Research and Data Domain as an intern. Through the internship, I learned not only about doing research to solve practical problems, but also developing my coding skills.

I wish to thank Dr. James S. Plank of University of Tennessee, and Dr. Alina Oprea and Mr. Kevin D. Bowers of RSA Laboratories. Parts of this work are the collaboration results with them.

I would also like to thank Dr. Hongwei Zhang, Dr. Nathan W. Fisher, and Dr. Cheng-Zhong Xu for serving on my dissertation defense committee.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1 Introduction

1.1 Introduction

Data reliability is a crucial issue to any commercial or scientific applications, which heavily depend on accessing data constantly to run businesses or conduct researches. In a computer system, data is stored on data storage systems, and then data reliability is essentially determined by the reliability of data storage systems. However, modern storage systems are complicated. A storage system is typically comprised of many components, from hardware to software, and a problem can occur in any component. When it happens, a storage system may stop working and needs time to be repaired. A worse consequence could be the stored data is permanently lost due to unrecoverable errors. This would be a disaster for companies offering online services, such as Google or Amazon, because such a failure may cause large revenue loss or even trust loss from customers. Therefore, it is critically important to build reliable storage systems to ensure data reliability.

A key technique to achieve high data reliability is by data redundancy. The basic idea is that for a piece of data, we first generate another piece of redundant data by a certain redundancy scheme. Then, we distribute the original data and the redundant data to multiple storage nodes. Note that storage node is an abstract concept, which could be a local hard disk or a remote storage server. After a period, if a failure happens to a storage node and prevents its stored data from accessing, we can reconstruct the data from other surviving storage nodes. Therefore, data redundancy provides the capability of fault tolerance. There are various redundancy schemes. A simple one is replication, which creates mirrors of the original data. More general term referring to redundancy scheme is *erasure code* [79], which defines mathematics means of computing redundant data.

A general erasure code can be represented by a notation of (n, k). A (n, k) code breaks a piece of user data into k symbols called *data* symbols, and encode it into n symbols by some mathematical means. If an erasure code is a systematical code [80], the k symbols of original data remain the same in the encoded n symbols, and we call the added $n - k$ symbols *parity* symbols. Systematical codes are mostly used in practice as there is no decoding cost when the part of user data is not lost. An interesting type of erasure codes is maximum distance separable (MDS) codes [80]. For a (n, k) MDS code, it can tolerate the loss of

any $n - k$ symbols without losing user data, and thus it imposes minimum storage overhead to achieve the same data reliability.

There are two categories of erasure codes that are mostly used in storage systems: XOR-based erasure codes and Reed-Solomon Codes [96]. XOR-based erasure codes perform only binary exclusive-or operation in both encoding and decoding, and hence they are efficient in terms of computation. This category of erasure codes include B-Code [117], X-Code [115], the WEAVER codes [47], EVENODD [20], RDP [31], and the STAR code [54]. However, these codes suffer from the restriction of (n, k) combinations. For example, EVENODD and RDP require $n-k = 2$. This constraint greatly limits the usage of XOR-based erasure codes. Another category is Reed-Solomon codes or its variants, such as Cauchy Reed-Solomon Codes [94]. Reed-Solomon codes uses those finite fields other than $GF(2)$ to perform encoding and decoding. Although the computation of Reed Solomon codes is not as fast as that of XOR-based codes, the code has two significant advantages. First, it provides flexible choices of (n, k), so that most environments can find their needed (n, k). Second, Reed-Solomon codes allows adding extra parity data when a system is running. For instance, if the original code used in a system is (n, k), it would be easy to extend the code to ($n + 1$, k) for the system. This property is particularly useful when data reliability needs to be dynamically enhanced.

In a reliable storage system, data redundancy can be implemented on different layers, which is described in Figure 1.1.

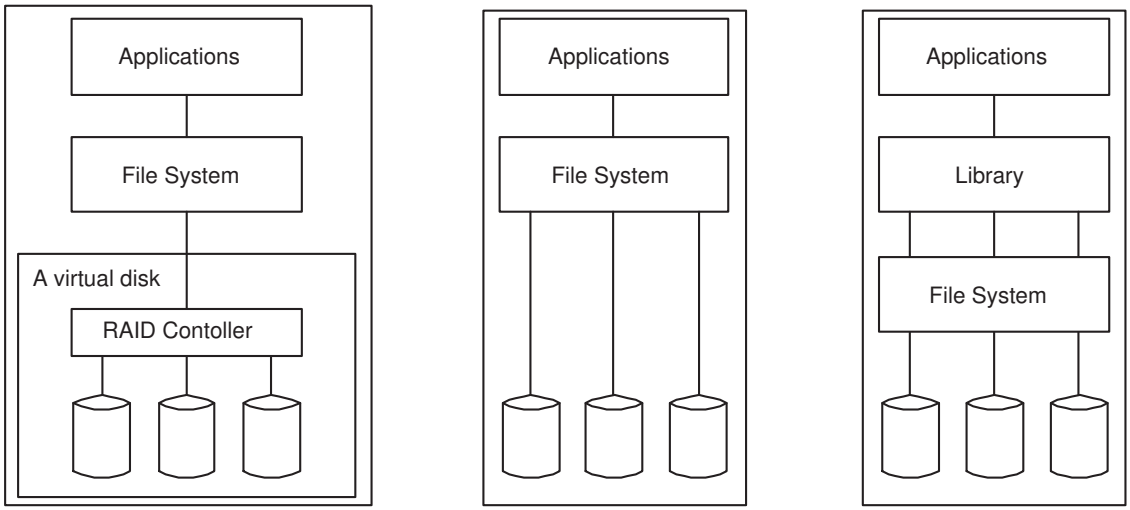


Figure 1.1: Data redundancy implementation layers

First, data redundancy can be implemented at hard disk layer. An example is Redundant Array of Independent Disks (RAID) [85]. A RAID system consists of a RAID controller and multiple hard disks. The overall RAID system is treated as a virtual single disk by operating systems. In the internal, the controller

serves all data access requests, including writing data and reading data. User data is then stored across multiple disks. When one disk or more fails to work, the failure event can be detected by the RAID controller. On one hand, the RAID controller continues providing normal data services by using other surviving disks; on the other hand, the controller restores the data stored in the failed disks at background. Therefore, RAID maintains its data service and data reliability. RAID has several categories. RAID-5 [85] can tolerate a single disk failure, while RAID-6 [20, 31] can tolerate up to two. Recent studies [70] have shown that RAID-6 is not able to provide needed data reliability as hard disk capacity grows dramatically, and it suggests that triple parity codes [54] should be used instead.

Besides hard disk layer, data redundancy can be implemented at file system layer. In this implementation, a reliable file system is like a RAID controller, which stores data across multiple storage nodes. If the storage nodes are local hard disks, the file system is also referred to as software RAID. However, the storage nodes can also be distributed on multiple sites. This can bring higher data reliability than RAID as it can tolerate a site failure. A common practice is that storage nodes are connected in a network. The network can be in a local area or in a wide area. During the system running, the client machines which are running user applications are connected to the network. When an application issues a data access request, the request will be received by the reliable file system. Then, the file system will communicate with storage servers to process it. After the process, the requested data will be returned to clients. The whole process is completely transparent to applications. Obviously, such an implementation makes the reliable system easily integrated into an existing system. Storage systems implemented in this way include HA-NFS [18], Zebra [52], Coda [97], Scotch [41], RAIF [65], HydraFS [106], PVFS [27], Panasas parallel file system [110], GlusterFS [43], and Lustre [58].

Besides file system layer, data redundancy can also be implemented at application layer. The difference from file system layer is that the data reliability services is provided by a library. Then, applications have to call a certain set of APIs to access the services. In the implementation of such a library, user data is also stripped to multiple storage nodes for high data reliability. The major benefit of this layer is simplicity. This layer does not need to provide full file system services, but only a portion of them, and hence the library can be developed quickly and easily. Furthermore, a wrapper implementation can be based on the library and work as a file system, and the data services can be supported by the file system. This can ease the integration of the library. Although the file system may not be POSIX compatible, it may be sufficient for a wide range of applications. The examples of storage systems implemented in this way are RAIN [25], Harp [72], HYDRAstor [33], Google File System (GFS) [40], and Hadoop [103].

The main topic of this dissertation is about how to efficiently implement a reliable file system by using erasures codes. We focus on several relevant problems. Firstly, as the use of erasure codes may cause significant performance overhead to storage systems, we studied how to improve both encoding and decoding performances for erasure codes. Then, we discussed how to leverage the inherent error correction capability of erasure codes to further improve data reliability for storage systems. Next, besides dealing with data reliability, we also presented an efficient algorithm for large finite fields so as to improve data security in a cloud storage system. Lastly and most importantly, we describe the design and implementation of a reliable file system, called *HyFS*. *HyFS* is capable of using general erasure codes and distributed storage servers to achieve high data reliability.

1.2 Contribution

The contributions of this dissertation contain five parts, which are summarized as follows.

This dissertation first proposes several algorithms to perform encoding operations efficiently for XOR-based erasure codes. When erasure codes are used in a storage system, encoding is an operation performed constantly when new data is written to the system, and hence it greatly impacts the storage system's performance. XOR-based erasure codes are described by equations that specify how parity symbols are calculated from data symbols, and existing implementations are constructed directly and naively from this specification. We found that the order of XOR operations is flexible and can greatly impact the encoding performance. Using this observation, we propose three new XOR-scheduling algorithms. The optimizations are based on understanding the semantics of erasure encoding and cannot be achieved by simple code transformations by a compiler.

An efficient decoding algorithm for RAID-6 erasure codes is presented. If a RAID system meets a disk failure, decoding operation has to be performed to reconstruct the failed data. One view of disk failures for RAID-6 is stop-and-fail error model, i.e., a disk either functions normally, or fails totally. This is called complete disk failure. However, some sectors or blocks of a disk can fail or corrupt because of various reasons, which is the so-called sector failures. As a RAID-6 system contains several disks, and each disk may encounter entire disk failure or sector failures, disk failures in a real system could become complicated. Then, we studied how to efficiently and uniformly decode all disk failure cases for RAID-6 codes. In theory, we first present the sufficient and necessary conditions to determine if a disk failure is recoverable or not by using Tanner graph. Practically, a universal decoding algorithm named *SCAN* is designed and evaluated.

This dissertation provides an efferent error correction algorithm for the STAR code. When data loss happens in a storage system, it can be either failure or silent error. Here a failure refers to a data loss with explicit error report from a disk drive. In contrast to failures, silent errors are less well-understood and not as sufficiently accounted for. Checksum at disk sector or block level is an effective and the most common technique. Unfortunately, checksum is not sufficient by itself. From a different perspective, however, redundancy is already in place introduced by error correcting codes to cope with failures at system level. Thus we advocate using error correcting codes to overcome both failures and silent errors simultaneously as a more unified and systematic mechanism. Specifically, we recommend the STAR code [54] as a very suitable candidate for such purpose. We propose an efficient error decoding algorithm named EEL (Efficient Error Locating) for the STAR code. The performance of EEL is evaluated against a naive one.

This dissertation introduces an efficient implementation of arithmetic operations for large finite fields. Most storage systems today employ erasure coding based on small finite fields (e.g., $GF(2^8)$ or $GF(2^{16})$) to provide fault tolerance in case of benign failures (for instance, drive crashes). They achieve efficiency through the use of small finite fields, but they have not been designed to sustain adversarial failures. With the advent of cloud storage, offered by providers such as Amazon S3 and others, a whole host of new failure models need to be considered, e.g., mis-configuration, insider threats, software bugs, and even natural calamities. Accordingly, storage systems have to be redesigned with robustness against adversarial failures. we provide efficient implementations of arithmetic operations for finite fields of characteristic two, ranging from $GF(2^{32})$ to $GF(2^{128})$. The main reason is that finite fields within this range are very suitable for secure data storage applications and systems. The new implementations achieve much more high performance than others.

Lastly and most importantly, this dissertation presents the design and implementation of a reliable file system, named *HyFS*, an essential component of a data storage system called *Hydra* [116]. *HyFS* is capable of employing general erasure codes and distributed storage servers to achieve high data reliability. When *HyFS* detects storage servers' failures, it will hide the failure from applications and continue its data service without any interruption. Then, at an appropriate time later, *HyFS* will automatically restore the data stored on the failed storage servers. As a result, data reliability is achieved. Furthermore, *HyFS* is implemented at file system layer so that the adoption of *HyFS* in a real system is easy and without any great effort. We believe that data reliability service is best to be implemented at file system layer and be full POSIX compatible. Regarding to data redundancy, *HyFS* supports a wide range of erasure codes, which can be simple replication scheme or complicated Reed-Solomon codes [96]. All these codes can be used simultaneously

in a *HyFS* system. Hence, *HyFS* can be easily configured to meet different data reliability requirements. For performance, a data request is served by multiple storage servers in parallel, which allows *HyFS* deliver high performance. Due to above reasons, we believe *HyFS* is an attractive solution for building reliable storage systems.

1.3 Organization

The rest of the dissertation is organized as follows. Chapter 2 discusses several efficient encoding algorithms for XOR-based erasure codes. Chapter 3 introduces an efficient decoding algorithm for RAID-6 erasure codes. Chapter 4 provides an efferent error correction algorithm for the STAR code. Chapter 5 presents an efficient implementation of arithmetic operations for large finite fields. Chapter 6 introduces a reliable file system called *HyFS*. Chapter 7 concludes the dissertation and outlines future research directions.

CHAPTER 2 Efficient Encoding for Erasure Codes

2.1 Introduction

As the amount of data increases exponentially in large distributed data storage systems, it is crucial to protect data from loss when storage devices fail to work. Recently, both academic and industrial storage systems have addressed this issue by relying on erasure codes to tolerate component failures. Examples include projects such as OceanStore [68], RAIF [65], and RAIN [25], and companies like Network Appliance [84], HP [114], IBM [47], Cleversafe [30], employing erasure codes such as RDP [31], the B-Code [117] and Reed-Solomon Codes [24, 96].

In an erasure coded system, a total of $n = k + m$ storage devices are employed, of which k hold data and m hold coding information. The act of *encoding* calculates the coding information from the data, and *decoding* reconstructs the data from surviving storage devices following one or more failures. Storage systems typically employ *Maximum Distance Separable (MDS)* codes [23], which ensure that the data can always be reconstructed as long as there are at least k storage devices that survive the failures.

Encoding is an operation performed constantly as new data is written to the system. Therefore, its performance is crucial to overall system performance. There are two classes of MDS codes – *Reed-Solomon* codes [96] and *XOR* codes (e.g. RDP [31] and X-code [115]). Although there are storage systems based on both types of codes, the XOR codes outperform the others significantly [92] and form the basis of most recent storage systems [25, 30, 65, 94].

This chapter addresses the issue of optimizing the encoding performance of XOR codes that are specifically tailored for data storage systems. XOR codes are described by equations that specify how the coding information is calculated from the data, and most implementations are constructed directly and naively from this specification. We call this naive implementation the *traditional* XOR-scheduling algorithm. The traditional algorithm is used in open source software such as Cleversafe's Information Dispersal implementation [30], Luby's Cauchy implementation [75], and Jerasure [89]. However, the order of XOR operations is flexible and can impact cache memory behavior of the codes significantly. Using this observation, we analyze the encoding process of XOR codes and propose three new XOR-scheduling algorithms. These algorithms employ different ways to improve the cache behavior of the XOR operations. The optimizations are based

on understanding the semantics of erasure encoding and cannot be achieved by simple code transformations by a compiler.

In this chapter, we focus on raw encoding performance just as in [92, 31]. We give a comprehensive demonstration of how two proposed new XOR-scheduling algorithms improve encoding performance on a variety of XOR codes and processing environments. Two of the new algorithms, called Data Words Guided (DWG) and Data Packets Guided (DGP), outperform the others. DWG improves the raw performance of encoding by 23% to 36% on various machines. Moreover, the improvement applies to all XOR codes and is therefore not code specific. DPG performs slightly worse than DWG scheduling, but its performance is more stable across different running environments. We perform a profiling analysis of our tests with the VTune Performance Analyzer [63] to illuminate the different cache effects that impact these scheduling algorithms.

2.2 Background

A storage system is composed of an array of n disks, each of which is the same size. Of these n disks, k hold data information and the remaining m hold coding information, often termed *parity*, which is calculated from the data. We label the data disks $D_0, \ldots, D_{k-1}$ and the parity disks $P_0, \ldots, P_{m-1}$. A typical system is pictured in Figure 2.1.

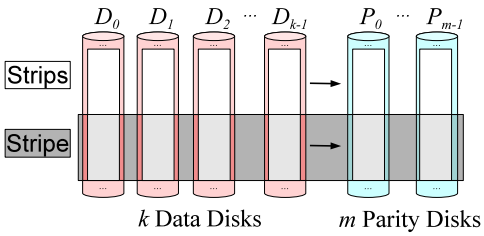


Figure 2.1: A typical storage system with erasure coding.

When encoding, one partitions each disk into strips of a fixed size. Each strip for a parity disk is encoded using one strip from each data disk, and the collection of $k + m$ strips is called a *stripe*. Thus, as in Figure 2.1, one may view each disk as a collection of strips, and one may view the entire system as a collection of stripes. The stripes are each encoded independently, and therefore if one desires to rotate the data and parity among the n disks for load balancing, one may do so by switching the disks' identities for each stripe.

Let us focus on a single stripe. Each XOR code has a parameter w, often termed the *word size* that further defines the code. This parameter is typically constrained by k and by the code. For example, for RDP [31], $w + 1$ must be a prime number, while for the X-codes [115] and Liberation codes [90], w must be a prime

number.

Each strip is partitioned into exactly w contiguous regions of bytes, called *packets*, labeled $D_{i,0}, \ldots D_{i,w-1}$ and $P_{j,0}, \ldots P_{j,w-1}$ for data and parity disks D_i and P_j respectively. Each packet is the same size, called the *packet size*. Strip sizes are therefore defined by the product of w and the packet size. An example of a stripe where $k = 4$, $m = 2$ and $w = 4$ is displayed in Figure 2.2.

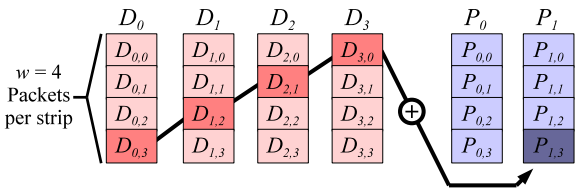


Figure 2.2: An example of one stripe where $k = 4$, $m = 2$ and $w = 4$.

For the purpose of defining a code, a packet size of one bit is convenient – parity bits are defined to be the XOR of collections of data bits. For example, in Figure 2.2, when the packet size is one, we can define the bit $P_{1,3}$ as being the XOR of bits $D_{0,3}$, $D_{1,2}$, $D_{2,1}$ and $D_{3,0}$, as it is in RDP [31]. However, for real implementations, packet sizes should be at least as big as a machine word, since CPUs can XOR two words in one machine instruction. Moreover, to improve CPU cache behavior, larger packet sizes are often preferable [92].

Codes are defined by specifying how each parity packet is constructed from the data packets. This may be done by listing equations, as in RDP [31], EVENODD [20], and X-code [115], or it may be done by employing a generator matrix [79]. To describe a code with a generator matrix, let us assume that the packet size is one bit. Therefore each data and parity strip is a w-bit word and their concatenation, which we label $(D|P)$, is a wn-bit word called the *codeword*. The generator matrix G has wk rows and wn columns and a specific format: $G = (I|H)$, where I is a $wk \times wk$ identity matrix. Encoding adheres to the following equation:

$$(D|P) = D * G = D * (I|H)$$

Thus, specifying the matrix H is sufficient to specify the code. Codes such as Liberation codes [90], Blaum-Roth codes [22] and Cauchy Reed-Solomon codes [24] are all specified in this manner.

Regardless of the specification technique, XOR codes boil down to lists of equations that construct parity packets as the XOR of collections of data packets. To be efficient, the total number of XOR operations should be minimized. Some codes, like RDP and X-code, achieve a lower bound of $(k - 1)$ XOR operations per

parity word, while others like Liberation codes, EVENODD and the STAR code are just above this lower bound [90, 92, 94].

2.2.1 Erasure Codes

We do not attempt to summarize all research concerning XOR codes. However, we detail the codes that are relevant to this chapter. We focus on MDS codes. Cauchy Reed-Solomon codes [24] are general-purpose XOR codes that may be defined for any values of k and m. The word size w is constrained such that $w \geq 2^n$, and the resulting generator matrix is typically dense, resulting in a large number of XOR operations for encoding. Plank and Xu describe how to produce sparser matrices [94], and these represent the best performing general-purpose erasure codes.

When m is constrained to equal two, the storage system is a RAID-6 system. There are many XOR codes designed for RAID-6, and these outperform Cauchy Reed-Solomon codes significantly. As stated above, RDP [31] achieves optimal encoding performance, and Liberation codes [90], Blaum-Roth codes [22] and EVENODD [20] are slightly worse. The STAR code extends EVENODD for $m = 3$ and like EVENODD greatly outperforms Cauchy Reed-Solomon coding. The X-codes [115] and B-Codes [117] are RAID-6 codes that also achieve optimal encoding performance, but require the parity information to distributed evenly across all $(k + m)$ storage devices, and are therefore less flexible than the others.

Both Huang [55] and Hafner [49] provide techniques for grouping common XOR operations in certain codes to reduce their number. These techniques are especially effective for decoding Liberation and Blaum-Roth codes. In contrast to this work, we do not improve performance by reducing the number of XOR operations, but instead by improving how the order of XOR operations affects cache behavior.

Finally, in 2007, Plank released an open source erasure coding library called Jerasure [89] which implements both Reed-Solomon and XOR erasure codes. The XOR codes must use a generator matrix specification, with Cauchy Reed-Solomon, Liberation and Blaum-Roth codes included as basic codes. The XOR reduction technique of Hafner [49] is included to improve the performance of decoding. The library is implemented in C and has demonstrated excellent performance compared to other open-source erasure coding implementations [92].

2.3 CPU Cache

All modern processors have at least one CPU cache that lies between the CPU and main memory. Its purpose is to bridge the performance gap between fast CPUs and relatively slow main memories [69]. Caches store recently referenced data, and when working effectively, reduce the number of accesses from CPU to main memory, each of which takes several instruction cycles and stalls the CPU. When an access to a piece of data is satisfied by the cache, it is called a cache *hit*; otherwise, it is called a cache *miss*. Many algorithms that optimize performance do so by reducing the number of cache misses.

There are three types of cache miss: *compulsory* miss, *capacity* miss, and *conflict* miss [53]. Compulsory misses happen when a piece of data is accessed for the first time. Capacity misses occur because of Least Recently Used (LRU) replacement policies on limited-size caches, and conflict misses occur in A-way associative caches when two pieces of data map to the same cache line.

To reduce cache misses, an algorithm needs to have enough data locality in time, space, or both. *Temporal* locality is when the time period between two consecutive accesses to the same data is very short. *Spatial* locality is when the space difference between the data in a series of accesses is very small. Good temporal locality reduces capacity misses, and good spatial locality reduces both compulsory and conflict misses.

Section 2.2 mentions that large packet sizes are desirable. This is to reduce compulsory misses: when the first byte of a packet is read into the cache, its following bytes are also read into cache because of standard cache prefetch mechanisms. Then, when these bytes are used sequentially, their compulsory misses are avoided.

In this chapter, we propose several algorithms to improve data locality for XOR-based erasure codes, so that encoding can be performed efficiently. We observed that encoding operation needs to access a lot of data when performing encoding computation. As the computation of XOR-based erasure codes involves only fast bitwise exclusive-or operation, how the encoding data is accessed now becomes an important performance factor. Furthermore, as we would demonstrate later, the data access order indeed greatly affects the encoding performance. We explored different data access orders for the same encoding computation, and we found that each order has its own benefits and disadvantages of CPU cache utilization. During the optimization, we used several well known techniques, such as loop transformation [69, 7, 109].

2.4 XOR-Scheduling Algorithms

We motivate our scheduling algorithms with a toy example erasure code for $k = 2, m = 2$, and $w = 2$. Although this code is simply a toy example, it is close to a real MDS code, namely, the EVENODD code. The code is described by Figure 2.3. In the figure, there are two data columns and two parity columns. A parity symbol with a shape is the bitwise XOR sum of data symbols with the same shape.

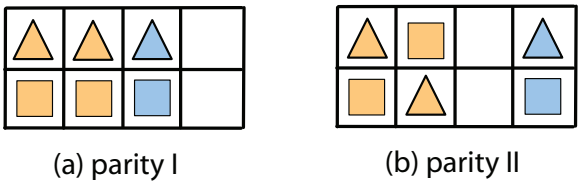


Figure 2.3: A toy example code construction.

Now, assume when implementing this code, the packet size is two machine words. Let the two words on one data packet $D_{[i,j]}$ (at row i and data column j) be $d_{[i,j],0}$ and $d_{[i,j],1}$, similarly for the parity words. The encoding of the entire system is in Table 2.1. Throughout this section, we will stick to this toy example to demonstrate how various XOR scheduling algorithms work.

$p_{[0,0],0} = d_{[0,0],0} + d_{[0,1],0}$
$p_{[0,0],1} = d_{[0,0],1} + d_{[0,1],1}$
$p_{[1,0],0} = d_{[1,0],0} + d_{[1,1],0}$
$p_{[1,0],1} = d_{[1,0],1} + d_{[1,1],1}$
$p_{[0,1],0} = d_{[0,0],0} + d_{[1,1],0}$
$p_{[0,1],1} = d_{[0,0],1} + d_{[1,1],1}$
$p_{[1,1],0} = d_{[1,0],0} + d_{[0,1],0}$
$p_{[1,1],1} = d_{[1,0],1} + d_{[0,1],1}$

Table 2.1: Encoding the toy example when the packet size is two.

2.4.1 Traditional XOR-scheduling Algorithm

Traditional XOR-scheduling performs the encoding in the order of how each parity symbol is calculated. In the toy example, the parity symbol $p_{[0,0]}$, $p_{[1,0]}$, $p_{[0,1]}$ and $p_{[1,1]}$ are performed in that order. To ease the explanation, we are going to assume that each parity word starts with a value of zero, and its calculation requires two XORs to update it. It will be clear how to remove this assumption later.

The schedule of XOR operations generated by the traditional algorithm on the toy example is listed in Table 2.2. Table 2.2 has two columns and each column contains multiple XOR operations. During the execution, the operations are performed in column order. Each XOR operation is followed by its ID. In this

case, the ID number indeed shows the execution order. Throughout our examples, the same XOR operation may appear in different positions in different scheduling algorithms, but its ID will remain constant.

parity column 0		parity column 1	
$p_{[0,0],0} += d_{[0,0],0}$	[1]	$p_{[0,1],0} += d_{[0,0],0}$	[9]
$p_{[0,0],1} += d_{[0,0],1}$	[2]	$p_{[0,1],1} += d_{[0,0],1}$	[10]
$p_{[0,0],0} += d_{[0,1],0}$	[3]	$p_{[0,1],0} += d_{[1,1],0}$	[11]
$p_{[0,0],1} += d_{[0,1],1}$	[4]	$p_{[0,1],1} += d_{[1,1],1}$	[12]
$p_{[1,0],0} += d_{[1,0],0}$	[5]	$p_{[1,1],0} += d_{[1,0],0}$	[13]
$p_{[1,0],1} += d_{[1,0],1}$	[6]	$p_{[1,1],1} += d_{[1,0],1}$	[14]
$p_{[1,0],0} += d_{[1,1],0}$	[7]	$p_{[1,1],0} += d_{[0,1],0}$	[15]
$p_{[1,0],1} += d_{[1,1],1}$	[8]	$p_{[1,1],1} += d_{[0,1],1}$	[16]

Table 2.2: The traditional algorithm on the toy example.

The first characteristic of the traditional algorithm is that it processes parity packets one by one, to completion. In the toy erasure code, there are four parity packets: $P_{[0,0]}$, $P_{[1,0]}$, $P_{[0,1]}$ and $P_{[1,1]}$. In the traditional algorithm, the contents of one parity packet will not be calculated until all of its previous packet is finished. The second characteristic of the traditional algorithm is that when calculating a parity packet, each related data packet is accessed in its entirety before the next data packet is accessed. In other words, the encoding proceeds packet by packet. For example, in Table 2.2, parity packet $P_{[0,0]}$ is computed from data packets $D_{[0,0]}$ and $D_{[0,1]}$, and all of $D_{[0,0]}$ is accessed before any of $D_{[0,1]}$. Thus, we summarize the characteristics of this algorithm as follows:

1. XOR operations are guided by the order of *parity* packets.

2. The innermost iteration is performed at the *packet* level.

Due to the above characteristics, this algorithm is termed Parity Packet Guided (*PPG*) XOR-scheduling. A pseudocode description of *PPG* XOR-scheduling is shown in Algorithm 1.

Algorithm 1 Parity Packet Guided (PPG) XOR-scheduling

INPUT: Data columns D, each one is a byte array
OUTPUT: Parity columns P, each one is a byte array

```
1: for each parity column P_j do
2:     for each parity packet P_{i,j} in column P_j do
3:         for each data packet D_{u,v} needed by P_{i,j} do
4:             for t=0; t<packet size; t++ do
5:                 p_{[i,j],t} += d_{[u,v],t};
6:             end for
7:         end for
8:     end for
9: end for
```

In Algorithm 1, the input is data columns and the output is parity columns. In practice, each column is one byte array, so that the data of one column can be written to one storage device by a single write request. The memory layout of the data is an important consideration on designing efficient XOR-scheduling algorithm. We will use the same input and output format for all algorithms introduced in this chapter.

Cache behavior analysis: *PPG* XOR-scheduling has good spatial locality because it accesses both the packets in one column and words in a packet sequentially.

2.4.2 Parity Words Guided (PWG) XOR-scheduling

A variant of *PPG* XOR-scheduling is Parity Words Guided (*PWG*) XOR-scheduling, and its characteristics are listed below:

1. XOR operations are guided by the order of *parity* packets.
2. The innermost iteration is performed at the *word* level.

The first characteristic of *PWG* XOR-scheduling is the same as that of *PPG* XOR-scheduling, and thus *parity* packets are also computed one by one in *PWG* XOR-scheduling. The two algorithms differ in their second characteristic. In *PWG* XOR-scheduling, the innermost iteration is at *word* level, while it is at *packet* level in *PPG* XOR-scheduling. Because of this reason, *PWG* XOR-scheduling generates a different schedule, shown in Table 2.3.

parity column 0		parity column 1	
$p_{[0,0],0} += d_{[0,0],0}$	[1]	$p_{[0,1],0} += d_{[0,0],0}$	[9]
$p_{[0,0],0} += d_{[0,1],0}$	[3]	$p_{[0,1],0} += d_{[1,1],0}$	[11]
$p_{[0,0],1} += d_{[0,0],1}$	[2]	$p_{[0,1],1} += d_{[0,0],1}$	[10]
$p_{[0,0],1} += d_{[0,1],1}$	[4]	$p_{[0,1],1} += d_{[1,1],1}$	[12]
$p_{[1,0],0} += d_{[1,0],0}$	[5]	$p_{[1,1],0} += d_{[1,0],0}$	[13]
$p_{[1,0],0} += d_{[1,1],0}$	[7]	$p_{[1,1],0} += d_{[0,1],0}$	[15]
$p_{[1,0],1} += d_{[1,0],1}$	[6]	$p_{[1,1],1} += d_{[1,0],1}$	[14]
$p_{[1,0],1} += d_{[1,1],1}$	[8]	$p_{[1,1],1} += d_{[0,1],1}$	[16]

Table 2.3: Parity Words Guided XOR-scheduling on the toy example.

In PWG, each parity word of a packet is calculated in its entirety before moving onto the next word. This is demonstrated in Table 2.3 where $p_{[0,0],0}$ is calculated before $p_{[0,0],1}$ is touched, resulting in the access sequence $\{p_{[0,0],0}, p_{[0,0],0}, p_{[0,0],1}, p_{[0,0],1}\}$ for parity packet $P_{[0,0]}$. A pseudocode description of this algorithm is shown in Algorithm 2.

Cache behavior analysis: *PWG* XOR-scheduling reverses the innermost two loops of *PPG* XOR-scheduling. On one hand, this loop transformation improves the temporal locality of $p_{[i,j],t}$ because now $p_{[i,j],t}$ is accessed

Algorithm 2 Parity Words Guided (PWG) XOR-scheduling

INPUT: Data columns D, each one is a byte array
OUTPUT: Parity columns P, each one is a byte array

```
1: for each parity column P_j do
2:   for each parity packet P_{i,j} in column P_j do
3:     for t=0; t<packet size; t++ do
4:       for each data packet D_{u,v} needed by P_{i,j} do
5:         p_{[i,j],t} += d_{[u,v],t};
6:       end for
7:     end for
8:   end for
9: end for
```

repeatedly in the innermost iteration (in line 5). On the other hand, it may lose the spatial locality of $d_{[u,v],t}$ because it accesses $d_{[u,v],t}$ in a less sequential way than *PPG* XOR-scheduling. Thus, the overall data locality may become worse since the loss of spatial locality could be greater than the gain of the temporal locality. This effect will be measured experimentally in Section 2.5.

2.4.3 Data Packets Guided (DPG) XOR-scheduling

The above two XOR scheduling algorithms (*PPG* and *PWG* XOR-scheduling) follow the intuitive idea that parity packets should be produced one by one. However, the schedule can be reordered so that the *data* packets are consumed one by one. An algorithm named Data Packets Guided (*DPG*) XOR-scheduling is based on this idea, and its characteristics are as follows:

1. XOR operations are guided by the order of *data* packets.

2. The innermost iteration is performed at data *packet* level.

The result of *DPG* XOR-scheduling on the toy example is shown in Table 2.4. Similar to Table 2.2 and 2.3, the XOR operations are executed in column order here

data column 0		data column 1	
$p_{[0,0],0} += d_{[0,0],0}$	[1]	$p_{[0,0],0} += d_{[0,1],0}$	[3]
$p_{[0,0],1} += d_{[0,0],1}$	[2]	$p_{[0,0],1} += d_{[0,1],1}$	[4]
$p_{[0,1],0} += d_{[0,0],0}$	[9]	$p_{[1,1],0} += d_{[0,1],0}$	[15]
$p_{[0,1],1} += d_{[0,0],1}$	[10]	$p_{[1,1],1} += d_{[0,1],1}$	[16]
$p_{[1,0],0} += d_{[1,0],0}$	[5]	$p_{[0,1],0} += d_{[1,1],0}$	[11]
$p_{[1,0],1} += d_{[1,0],1}$	[6]	$p_{[0,1],1} += d_{[1,1],1}$	[12]
$p_{[1,1],0} += d_{[1,0],0}$	[13]	$p_{[1,0],0} += d_{[1,1],0}$	[7]
$p_{[1,1],1} += d_{[1,0],1}$	[14]	$p_{[1,0],1} += d_{[1,1],1}$	[8]

Table 2.4: Data Packets Guided XOR-scheduling on the toy example.

This algorithm is similar to *PPG* XOR-scheduling in that their innermost iterations are both performed at *packet* level. Their difference is in type of packets guiding the XOR operations: in *PPG* XOR-scheduling, it is the *parity* packets; while in this algorithm, it is the *data* packets. Therefore, Table 2.4 shows that the contents of $D_{[1,0]}$ will not participate in computation until all of $D_{[0,0]}$ is finished, and when $D_{[0,0]}$ is processed, the access sequence for it is $\{d_{[0,0],0}, d_{[0,0],1}, d_{[0,0],0}, d_{[0,0],1}\}$, where $\{d_{[0,0],0}, d_{[0,0],1}\}$ is followed by another $\{d_{[0,0],0}, d_{[0,0],1}\}$. A pseudocode description of this algorithm is shown in Algorithm 3.

Algorithm 3 Data Packets Guided (DPG) XOR-scheduling

INPUT: Data columns D, each one is a byte array
OUTPUT: Parity columns P, each one is a byte array

```
1: for each data column D_j do
2:   for each data packet D_{i,j} in column D_j do
3:     for each parity packet P_{u,v} that needs D_{i,j} do
4:       for t=0; t<packet size; t++ do
5:         p_{[u,v],t} += d_{[i,j],t};
6:       end for
7:     end for
8:   end for
9: end for
```

Cache behavior analysis: Different from *PPG* XOR-scheduling which follows directly how parity symbols are constructed, *DPG* XOR-scheduling calculates parity symbols from another perspective based on the participation order of data symbols in encoding. The purpose is to improve the locality of data symbols. Now, in *DPG* XOR-scheduling, the outermost loop is on data columns, and the access to data packets is in a more sequential way than that of *PPG* XOR-scheduling. In a typical RAID-6 system, $m = 2$ and $k \geq 10$, so m is much smaller than k. Thus, the locality of data packets is more important than that of parity packets. Hence, this transformation should improve the overall data locality of *DPG* and may lead to higher encoding performance.

2.4.4 Data Words Guided (DWG) XOR-scheduling

Following the approach of constructing *PWG* XOR-scheduling from *PPG* XOR-scheduling, an variant of *DPG* XOR-scheduling is developed. We call the variant Data Words Guided (*DWG*) XOR-scheduling, and its characteristics are listed below:

1. XOR operations are guided by the order of *data* packets.

2. The innermost iteration is performed at data *word* level.

data column 0		data column 1	
$p_{[0,0],0} += d_{[0,0],0}$	[1]	$p_{[0,0],0} += d_{[0,1],0}$	[3]
$p_{[0,1],0} += d_{[0,0],0}$	[9]	$p_{[1,1],0} += d_{[0,1],0}$	[15]
$p_{[0,0],1} += d_{[0,0],1}$	[2]	$p_{[0,0],1} += d_{[0,1],1}$	[4]
$p_{[0,1],1} += d_{[0,0],1}$	[10]	$p_{[1,1],1} += d_{[0,1],1}$	[16]
$p_{[1,0],0} += d_{[1,0],0}$	[5]	$p_{[0,1],0} += d_{[1,1],0}$	[11]
$p_{[1,1],0} += d_{[1,0],0}$	[13]	$p_{[1,0],0} += d_{[1,1],0}$	[7]
$p_{[1,0],1} += d_{[1,0],1}$	[6]	$p_{[0,1],1} += d_{[1,1],1}$	[12]
$p_{[1,1],1} += d_{[1,0],1}$	[14]	$p_{[1,0],1} += d_{[1,1],1}$	[8]

Table 2.5: Data Words Guided XOR-scheduling on the toy example.

The result of *DWG* XOR-scheduling on the toy example is shown in Table 2.5.

In Table 2.5, all of the equations involving data packet $D_{[0,0]}$ appear before those involving $D_{[1,0]}$, following the first characteristic. By the second characteristic, each data word is used for all parity calculations before moving onto the next data word in the same packet. This is shown in Table 2.5 where $d_{[0,0],0}$ is used to calculate both $p_{[0,0],0}$ and $p_{[0,1],0}$ before $d_{[0,0],1}$ is touched. Then, the access sequence for $D_{[0,0]}$ is $\{d_{[0,0],0}, d_{[0,0],0}, d_{[0,0],1}, d_{[0,0],1}\}$, where all the access of $d_{[0,0],0}$ appears ahead of $d_{[0,0],1}$. A pseudocode description of this algorithm is shown in Algorithm 4.

Algorithm 4 Data Words Guided (DWG) XOR-scheduling

INPUT: Data columns D, each one is a byte array
OUTPUT: Parity columns P, each one is a byte array

```
1: for each data column D_j do
2:   for each data packet D_{i,j} in column D_j do
3:     for t=0; t<packet size; t++ do
4:       for each parity packet P_{u,v} that needs D_{i,j} do
5:         p_{[u,v],t} += d_{[i,j],t};
6:       end for
7:     end for
8:   end for
9: end for
```

Cache behavior analysis: *DWG* XOR-scheduling differs with *DPG* XOR-scheduling in the conversion of the innermost two loops. This transformation is to improve temporal locality of $d_{[i,j],t}$. Note that *DWG* XOR-scheduling may reduce some spatial locality of $p_{[u,v],t}$ and $d_{[i,j],t}$, but our observation is that the gain of temporal locality of $d_{[i,j],t}$ brought by the transformation is usually greater than the loss of spatial locality, and thus the overall data locality is better. Again, this is because in most storage systems, m is a small value, such as $m = 2$ in RAID-6 systems, and most data and parity words will remain in the cache across iterations of the innermost loop (in line 4 of Algorithm 4).

It is further worth noting that these loop transformations employed in the four algorithms cannot be simply achieved by a compiler. There are three reasons contributing to that:

1. Each algorithm contains four nested loops. Such a complicated loop structure makes it hard for a compiler to optimize if possible.

2. For easy explanation, the presented pseudocode is a simplified version of its source code. In fact, a real implementation is filled with many more details. The details may obscure the loop structure and complicate the optimization of a compiler.

3. Although Algorithm 3 and 4 look like Algorithm 1 and 2, they are significantly different: for a given data packet, Algorithm 3 and 4 need to identify the parity packets it is involved; however, Algorithm 1 and 2 have to find the data packets a particular parity packet is computed from. Obviously, they are two very different functions. A compiler has no way to derive one from another.

Our observation is also consistent with the results from Lebeck et al. [69], who found that manual optimization is still needed to improve an algorithm's performance.

2.4.5 Implementation Details

Generating and using schedules that are as detailed as those in Tables 2.2-2.5 takes too much space, but is not necessary because a list of each data packet's associated parity packets may be constructed simply from the packet's row of the generator matrix. Once that list is generated, it may be used for every word in the packet.

Another important detail of the *DPG* and *DWG* XOR-schedulings is how to initialize parity packets. They cannot be simply initialized to zero, since that increases the number of XOR operations. Instead, we copy the first data packets that are used for each parity packet, instead of XOR-ing them. Implementationally, this is simple in the *PPG* and *PWG* XOR-scheduling, where the first data packets can be easily identified at the beginning. In the *DPG* and *DWG* XOR-scheduling, it is more complex, since for a data packet, we need to know whether it is the first data packet of one parity packet. Fortunately, all the codes addressed in this chapter have regular structures which makes this determination straightforward.

A final detail involves the codes that have extra information in addition to their generator matrices—EVENODD, the STAR code, and RDP. EVENODD employs a temporary packet S, which must be calculated as an intermediate sum for every packet in parity column P_1. To handle this, all the XOR-scheduling algorithms perform two passes. In the first pass, the data packets are used to calculate P_0, P_1 and S, and then

a second pass XOR's S with the packets of P_1. The STAR code, an extension of EVENODD, is handled similarly. In RDP, all but one of the packets in P_1 are calculated using packets in P_0 in addition to data packets. For that reason, two passes are also performed in RDP: a first one that calculates P_0 and P_1 without the P_0 packets, and a second one that XOR's the P_0 packets into P_1.

2.5 Performance Evaluation

To compare the four XOR-scheduling algorithms, we have conducted experiments that apply each algorithm to many popular XOR-based erasure codes: the Liberation codes [90], EVENODD [20], RDP [31], X-code [115], and the STAR code [54]. All are RAID-6 codes ($m = 2$), with the exception of the STAR code which tolerates three failures ($m = 3$).

We have conducted two sets of experiments. The first set is from a system practitioner's point of view. We treat CPU cache as a black box and we measure the encoding performance without examining the detailed interactions of the scheduling algorithms with caches. The corresponding results are presented in Sections 2.5.2 and 2.5.4. The second set is to further examine the interactions of the scheduling algorithms with the memory hierarchy of test machines. This verifies that the encoding performance gains of the scheduling algorithms do indeed come from their effective use of CPU caches. Instead of simulations or simplistic modeling, we choose to perform profiling CPU events to truly reflect real cache behavior. The profiling results are presented in Sec. 2.5.3.

2.5.1 Experiment Setup

Since CPU cache behavior is complicated, we have run experiments on various machines to get a comprehensive view of all algorithms. The experiments in this section use four different machines, all with Intel processors. The configuration of these machines is shown in Table 2.6.

Machine	CPU Model	Speed	L1 Cache	L2 Cache	Memory	gcc
P4	Pentium 4	3.0GHz	28KB	2MB	DDR 533MHz	4.4.1
Pd	Pentium Dual Core	2.8GHz	$2 \cdot 16$KB	$2 \cdot 1$MB	DDR2 533MHz	4.2.1
Pc2d	Pentium Core 2 Duo	2.1GHz	$2 \cdot 32$KB	3MB	DDR2 667MHz	4.1.3
Pc2q	Pentium Core 2 Quad	2.4GHz	$4 \cdot 32$KB	$2 \cdot 4$MB	DDR2 800MHz	4.3.2

Table 2.6: Test machine configurations.

All machines run 64-bit processors, and they are installed with 64-bit version of Linux. Since 64-bit machines perform XOR operations on 64-bit words, their encoding performance is as much as a factor of

two faster than their 32-bit counterparts [92]. The implementations of all algorithms do not use complicated performance enhancement techniques, such as SIMD (single instruction multiple data) instructions [60] or software prefetching [74]. These techniques may speed up XOR operations or improve CPU cache behavior, and they are currently being explored by other researchers.

Our source code is written in C and compiled using **gcc**. We tried both **-O2** and **-O3** optimization flags, of which **-O2** is recommended for most applications while **-O3** is the highest level of optimization [39]. No other optimization flags are set, such as prefetch-loop-arrays [83]. Since the performance results of them are similar, we only present the results for **-O2** here. Each machine's **gcc** version is listed in the last column of Table 2.6. As single-threaded program is easy to implement and optimize, the code runs only on one thread, and thus does not take advantage of multiple cores.

For the Liberation codes, we use the Jerasure open source coding library [89] as the implementation for *PPG* XOR-scheduling, since that is exactly what Jerasure implements. For the other codes and algorithms, we used Jerasure as our base and crafted custom code from it.

Our experimental framework is very similar to that in [92]. All tests are performed in main memory, without actual disk I/O, because that introduces a great deal of variability. We encode totally a gigabyte of data, which is randomly generated. We evaluate encoding performance using the metric of *encoding throughput*, calculated with the equation below:

$$\text{Encoding throughput} = \frac{\text{Total user data size}}{\text{Encoding time}}$$

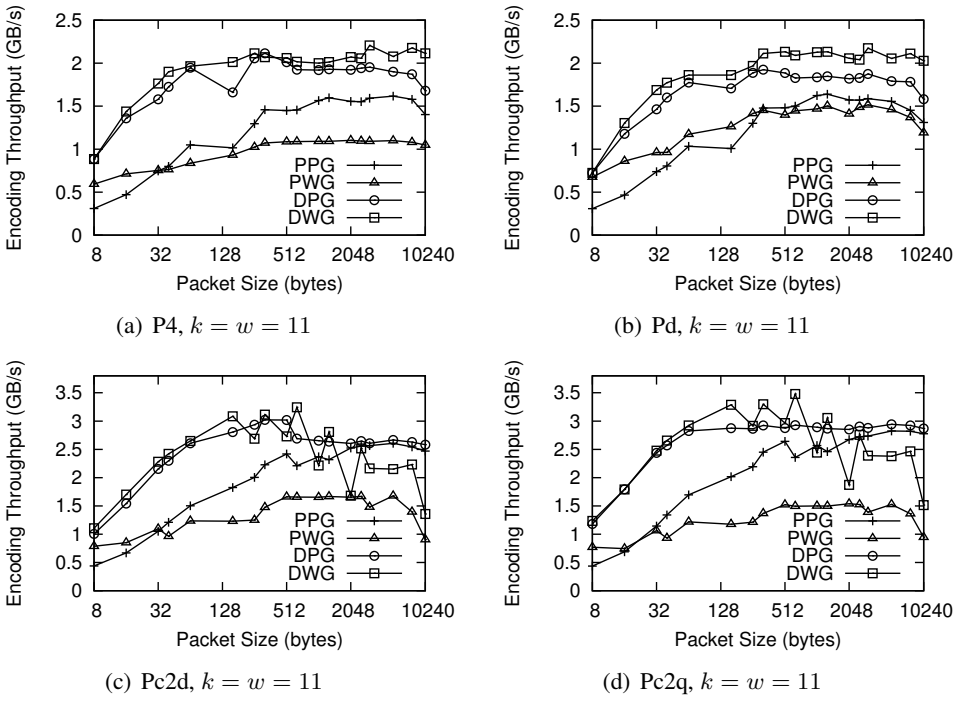
Encoding throughput represents how quickly one can turn user data into parity data with a given code, algorithm and machine. The encoding time is calculated by the **gettimeofday()** system call. Each data point is the average of 30 test runs. The data plotted is all within a confidence interval of 95%.

2.5.2 Experiments on the Liberation Codes

This section focuses on a performance comparison of the Liberation codes, because it is the one code for which an open source implementation is available.

Parameters $k = w = 11, m = 2$

The parameters of the first experiment for the Liberation code are $k = w = 11$, and $m = 2$, as these represent a typical RAID-6 system. We vary the packet size because it can impact performance greatly [92].



(a) P4, $k = w = 11$ (b) Pd, $k = w = 11$

(c) Pc2d, $k = w = 11$ (d) Pc2q, $k = w = 11$

Figure 2.4: Encoding performance of Liberation codes, $k = w = 11$ and $m = 2$.

Figure 2.4(a) compares the algorithms' performance on machine P4. The results can be summarized as follows:

1. As packet sizes increase from a small value, the performance of all scheduling algorithms improves significantly. This mirrors the observations by Plank et al. [92].

2. For all packet sizes, *DWG* and *DPG* XOR-scheduling achieve much better encoding performance than the other two algorithms. It is because *DWG* and *DPG* XOR-scheduling have better data locality than the other two, matching the analysis given in Section 2.4.

3. *DWG* XOR-scheduling has the highest peak performance, and *DPG* XOR-scheduling performs second best. *PPG* XOR-scheduling is the next one, and *PWG* XOR-scheduling is the last one.

4. *DWG* XOR-scheduling and *DPG* XOR-scheduling achieve their peak encoding performance with a relatively small packet size. This is useful because it provides real systems more flexibility on choosing packet sizes for high performance.

Figure 2.4(b) displays the results on the Pd machine. The performance is very similar to P4, so the four above observations on P4 also apply to Pd.

Figure 2.4(c) shows the results for machine Pc2d. This machine shows a new performance feature—after reaching the peak performance with a small packet size, the performance of *DWG* XOR-scheduling

becomes unstable, and at some points it performs worse than *DPG* and *PPG* XOR-scheduling. This will be analyzed further in Section 2.5.3. Nonetheless, the two most important observations still hold: 1) the peak performance of *DWG* XOR-scheduling is higher than all others; 2) the peak performance is obtained with small packet size. Compared to *DWG* XOR-scheduling, although *DPG* XOR-scheduling has lower peak performance, it achieves more stable performance and its performance is always higher than that of *PPG* and *PWG* XOR-scheduling.

Figure 2.4(d) shows the results on machine Pc2q. These are very similar to Pc2d, and the observations for Pc2d are also valid for Pc2q.

The peak performance of *DWG* and *PPG* XOR-scheduling (the traditional one) on the four machines are summarized in Table 2.7. It clearly displays that *DWG* XOR-scheduling achieves significant performance improvement over *PPG* XOR-scheduling, from 23% to 36%.

Machine	DWG	PPG	Improvement
P4	2.207 GB/s	1.616 GB/s	36%
Pd	2.174	1.64	32%
Pc2d	3.243	2.607	24%
Pc2q	3.479	2.824	23%

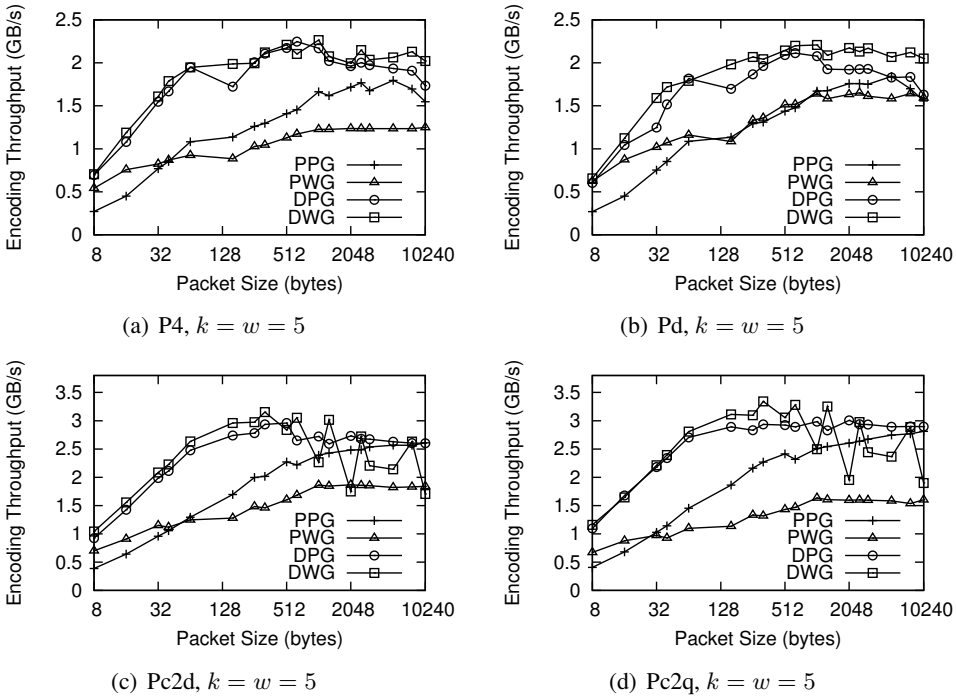
Table 2.7: Comparison of peak encoding performance of Liberation codes for $k = w = 11$ and $m = 2$.

Modifying k and w

In the Liberation codes, the number of parity devices is fixed at two. However, systems may range in size from small k to large.

To observe potential sensitivity to the number of data devices k and the number of packets per stripe w, Figure 2.5 and Figure 2.6 display the encoding performance of the Liberation codes for $k = w = 5$ and $k = w = 17$. The results mirror the results for $k = 11$, and the observations that held for $k = 11$ also hold for the smaller and larger values of k. Thus, all scheduling algorithms are stable for this code among this range of parameters.

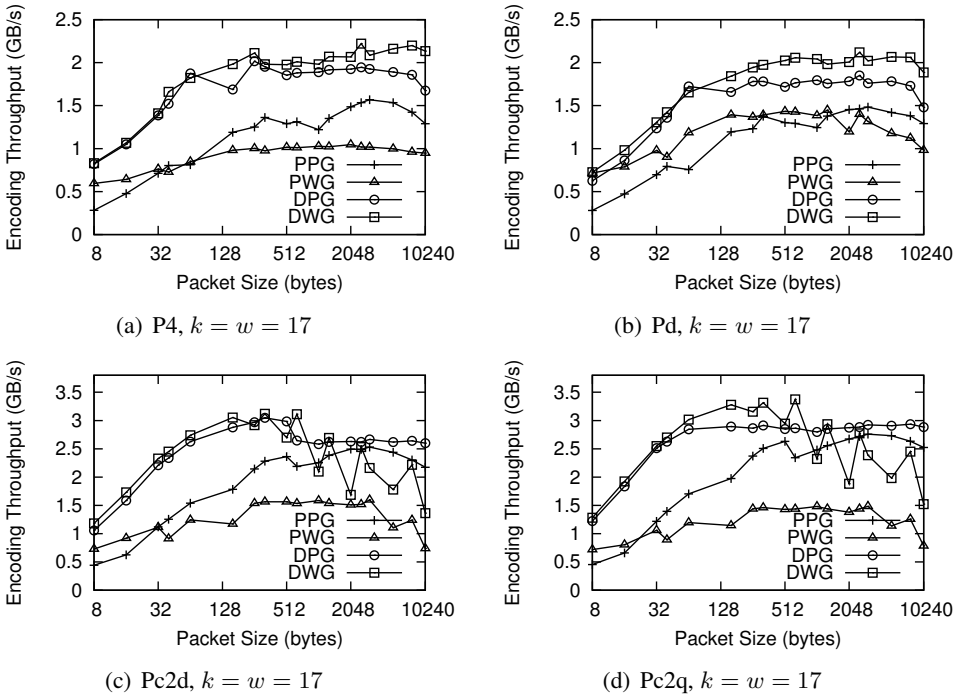
It is worth noting that the values of k are prime numbers in above experiments. In practice, this is not required. A well known technique called shortening [79, 94] can be applied to produce a shorter code but with a certain efficiency loss in space utilization.



(a) P4, $k = w = 5$ (b) Pd, $k = w = 5$

(c) Pc2d, $k = w = 5$ (d) Pc2q, $k = w = 5$

Figure 2.5: Encoding performance of Liberation codes for $k = w = 5$ and $m = 2$.



(a) P4, $k = w = 17$ (b) Pd, $k = w = 17$

(c) Pc2d, $k = w = 17$ (d) Pc2q, $k = w = 17$

Figure 2.6: Encoding performance of Liberation codes for $k = w = 17$ and $m = 2$.

2.5.3 Performance Profiling for the Liberation Code

To better understand the interactions of XOR-scheduling algorithms and the memory hierarchy of test machines, we performed a profiling experiment with the Liberation Codes with $k = w = 11$. We used the Intel VTune performance Analyzer [63], which leverages performance counting registers available on certain Intel processors. The event-based sampling (EBS) profiler of VTune allows one to register certain events for monitoring, and then VTune samples the number of these events during live runs of programs on the processor. For each event, one specifies a *sample-after-value (SAV)*. VTune maintains a hardware counter of the event, and when the counter reaches the SAV value, VTune increments a user-specified value and resets the counter. Thus, one may trade off the precision of sampling with the invasiveness of VTune.

This profiling approach differs from those based on simulation, such as Cachegrind [26]. Its main advantage is that it profiles real executions on the processors, and thus is not limited by assumptions made by a simulator. Its main disadvantage is the tradeoff between sampling granularity and profiling invasiveness. A second disadvantage is the fact that while one can count certain events, it may be hard to interpret why those events are happening. Regardless, the various event counts can give us insight into how a program is behaving.

For the XOR-scheduling algorithms, we chose to monitor the events which we summarize below. One measures instruction counts, while the remaining three assess different parts of the memory hierarchy. For complete descriptions, please see the Intel software developer's manual [61].

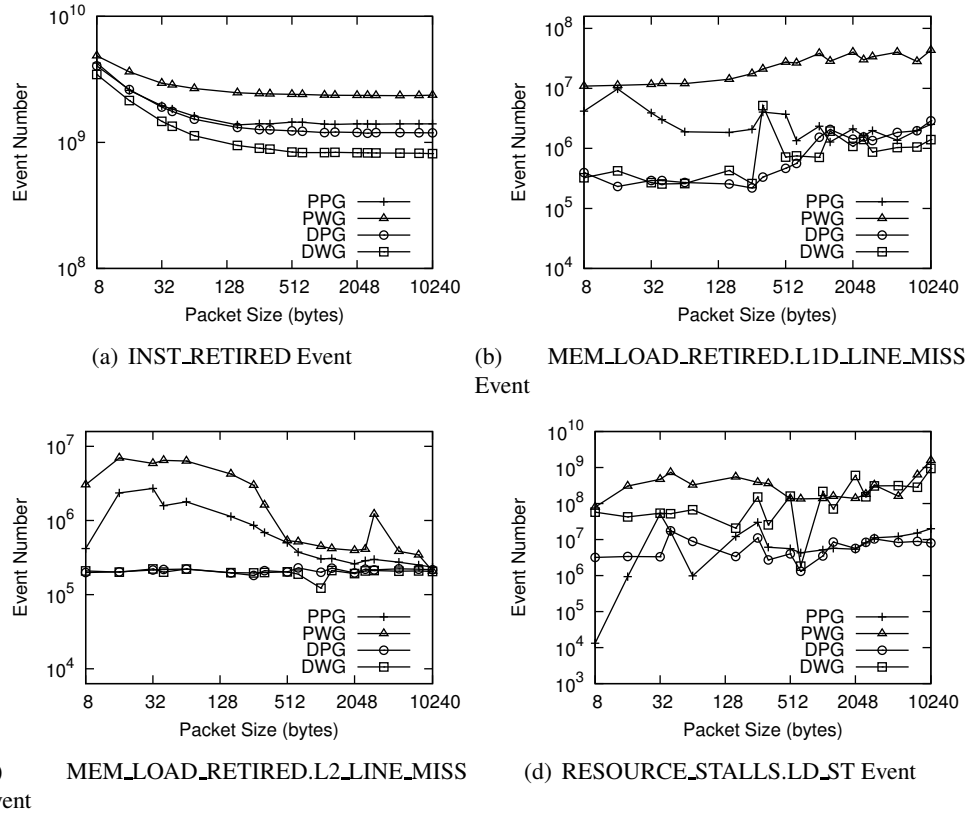
- **INST_RETIRED**: This counts the number of instructions that are retired. On processors with speculative execution, this is a measure of the instructions that have completed successfully and had their results written to the cache.

- **MEM_LOAD_RETIRED.L1D_LINE_MISS**: This is a measure of how many load operations miss the L1 data cache and send a request to the L2 cache to satisfy the load operation.

- **MEM_LOAD_RETIRED.L2_LINE_MISS**: This is a measure of how many L2 cache requests are unsatisfied and must be reloaded from main memory.

- **RESOURCE_STALLS.LD_ST**: This is an event when the processor stalls because the load and store buffers used for pipelined and out-of-order execution become full.

For the INST_RETIRED event, we set the SAV value to 100,000. For all others, we set it to 10,000.



(a) INST_RETIRED Event

(b) MEM_LOAD_RETIRED.L1D_LINE_MISS Event

(c) MEM_LOAD_RETIRED.L2_LINE_MISS Event

(d) RESOURCE_STALLS.LD_ST Event

Figure 2.7: Performance profiling results of Liberation codes for $k = w = 11$ and $m = 2$ on machine Pc2q.

We show the results of the profiling tests in Figure 2.7. The machines Pc2d and Pc2q support the profiling of the above events. Their profiling results are similar, so we only display Pc2q in Figure 2.7. It is worth noting that one cannot derive actual performance from the various event counts. Instead, the profiling helps us to gain insight into why the performance of the various algorithms is as it is. We evaluate each of the events below.

INST_RETIRED (Figure 2.7(a)): For all algorithms, as the packet size increases, the instruction counts decrease, becoming roughly constant for packet sizes greater than 512 bytes. This is because fewer loop iterations are required as the packet size increases. Of the four algorithms, the *PWG* algorithm retires the most instructions and *DWG* the least. The other two exhibit similar performance.

MEM_LOAD_RETIRED.L1D_LINE_MISS (Figure 2.7(b)): This measures the number of processor stalls due to L1 cache misses. As the packet sizes increase, these values increase too, because fewer packets fit into the L1 cache. The *PWG* algorithm displays significantly more stalls due to L1 cache misses than the others, which exhibit similar numbers as the packet sizes grow to 1K and beyond.

MEM_LOAD_RETIRED.L2_LINE_MISS (Figure 2.7(c)): This measures the number of processor stalls due to L2 cache misses. Since these misses must be satisfied from main memory, the impact of these

stalls is greater than for L1 misses. The *DWG* and *DPG* algorithms exhibit stable performance with respect to L2 cache misses, and this performance is independent of the packet size. Moreover, they are better than the *PPG* and *PWG* algorithms for all packet sizes. This is the main contributing factor to the *DWG* and *DPG* algorithms' superior performance in Figure 2.4(d). The *PPG* and *PWG* algorithms have fewer L2 cache misses as their packet sizes increase.

RESOURCE_STALLS.LD_ST (Figure 2.7(d)): In this figure, the *PPG* and *DPG* algorithms exhibit the fewest stalls due to the load and store buffers being full. The *DWG* algorithm shows the greatest variability here, especially for large packet sizes. This is reflected in its variable overall performance in Figure 2.4(d). As in the other figures, the *PWG* algorithm exhibits the worst overall performance.

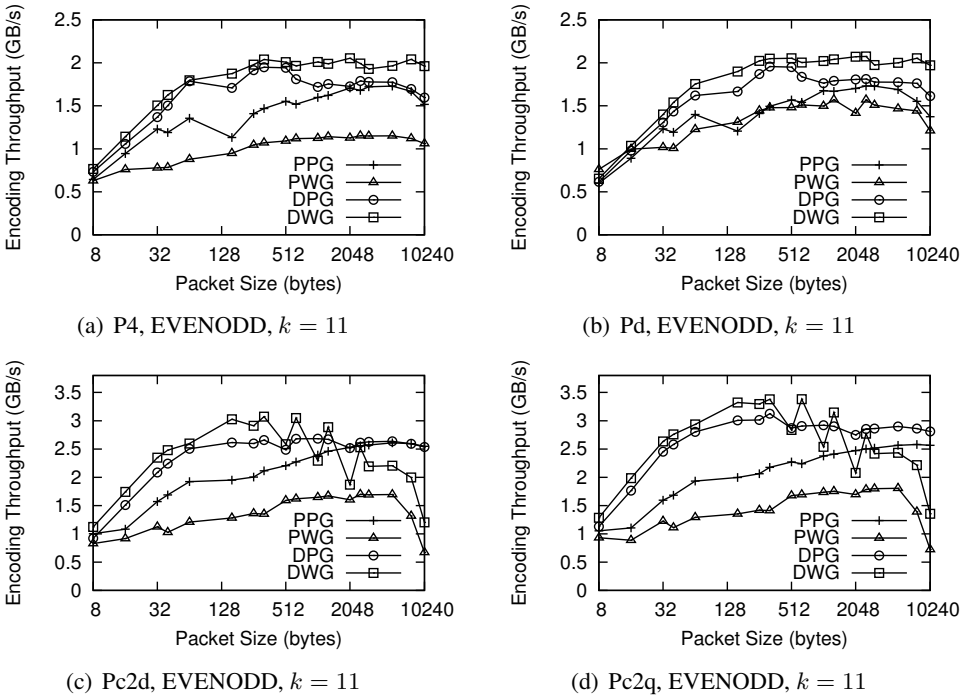
In summary, while the *DWG* algorithm achieves the best performance for each code and machine, it is more sensitive to having the packet size impact the cache behavior. This is most pronounced in Figures 2.7(b)) and 2.7(d)). On the other hand, the *DPG* algorithm achieves a nicer blend of performance and stability. The *PWG* algorithm achieves the worst performance, as reflected in every event measured in our profiling experiments. It is worth noting that the *DPG* and *DWG* algorithms have fewer L1 and L2 cache misses at lower packet sizes, and thus both algorithms achieve their peak performance at smaller packet sizes than *PPG* and *PWG*.

It is interesting to observe that although *DPG* and *DWG* were originally designed to reduce the number of cache misses, the profiling results show that they also effectively decrease the number of executed instructions. Regardless, it is significant to note that the scheduling of XOR operations indeed affects all of these performance events.

Finally, we did profile the other erasure codes that are discussed below. However, since their results were very similar to the Liberation code profiling, we omit their presentation here.

2.5.4 Experiments on Other Erasure Codes

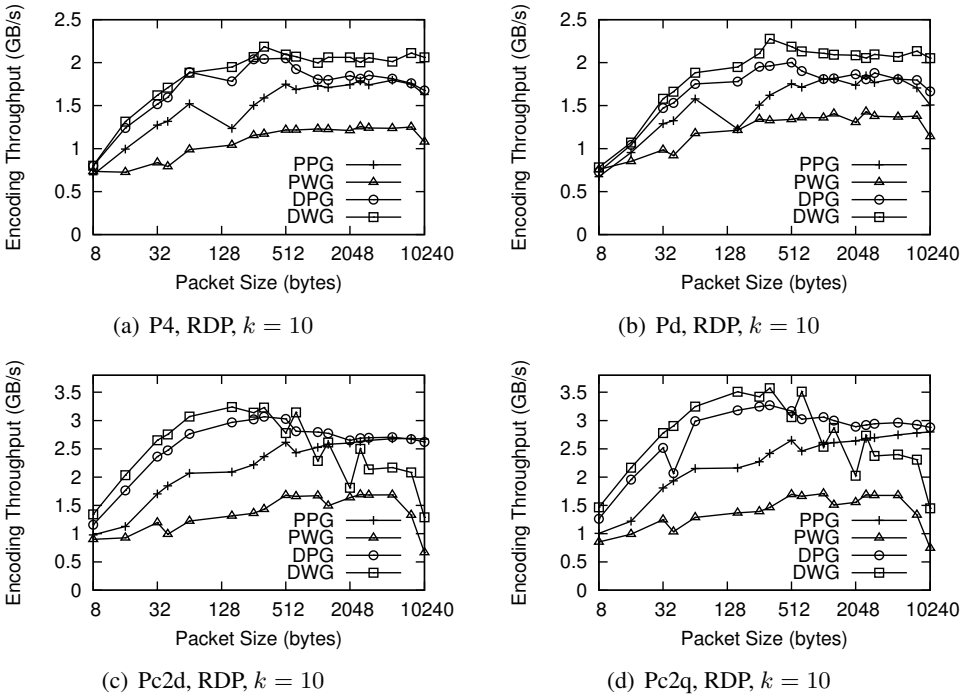
To compare the algorithms on other codes, this section tests EVENODD, RDP, X-code, and the STAR code. Again, all but the STAR code are RAID-6 codes. Among them, EVENODD and RDP have efficient decoding algorithms, while RDP has better encoding performance and EVENODD has better update performance [94]. X-code has good encoding/decoding/update performance; however, it is a vertical code and has limitations on the choice of the value of k. Lastly, the STAR code can tolerate up to 3 disk failures. These various codes impose different constraints on k and w, so we selected values that would match most closely with the Liberation codes example. The values are summarized in Table 2.8.



(a) P4, EVENODD, $k = 11$ (b) Pd, EVENODD, $k = 11$

(c) Pc2d, EVENODD, $k = 11$ (d) Pc2q, EVENODD, $k = 11$

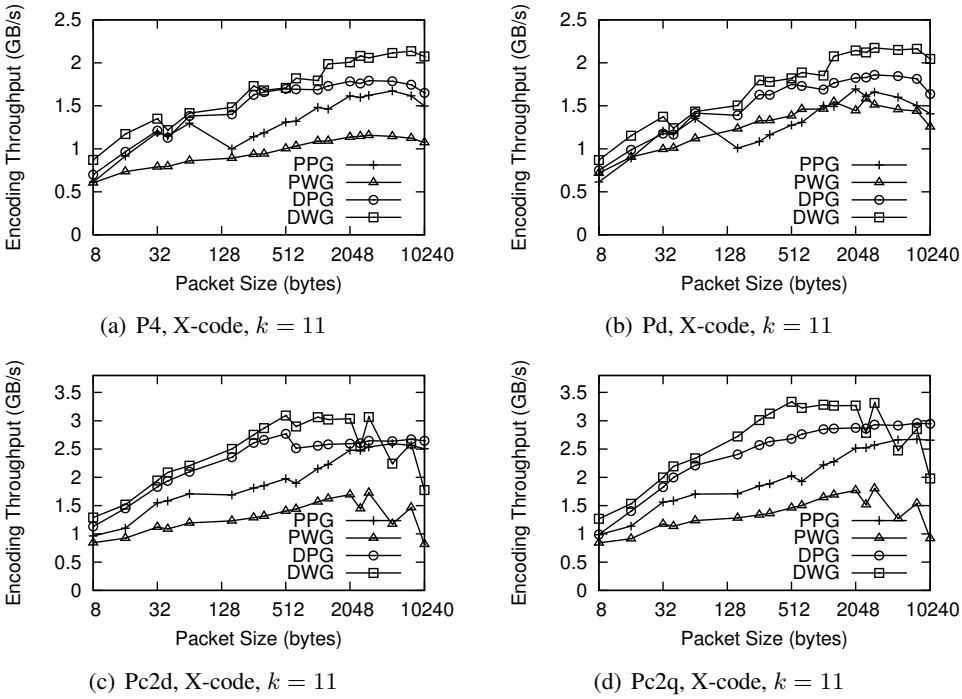
Figure 2.8: Encoding performance of EVENODD for $k = 11,\ w = 10$ and $m = 2$.



(a) P4, RDP, $k = 10$ (b) Pd, RDP, $k = 10$

(c) Pc2d, RDP, $k = 10$ (d) Pc2q, RDP, $k = 10$

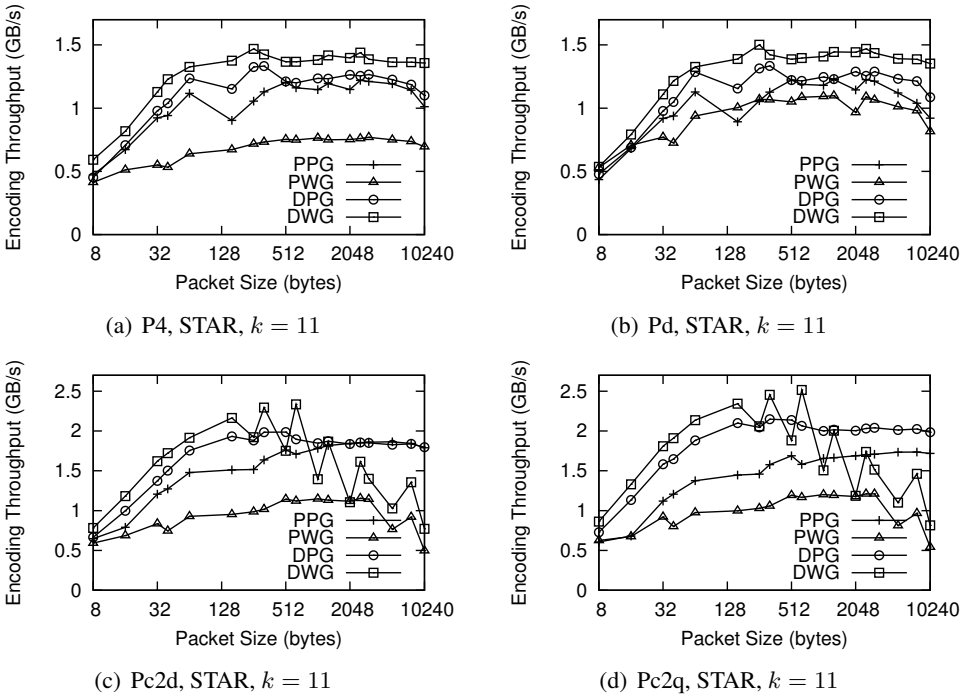
Figure 2.9: Encoding performance of RDP for $k = w = 10$ and $m = 2$.



(a) P4, X-code, $k = 11$ (b) Pd, X-code, $k = 11$

(c) Pc2d, X-code, $k = 11$ (d) Pc2q, X-code, $k = 11$

Figure 2.10: Encoding performance of X-code for $k = 11,\ w = 13$, and $m = 2$.



(a) P4, STAR, $k = 11$ (b) Pd, STAR, $k = 11$

(c) Pc2d, STAR, $k = 11$ (d) Pc2q, STAR, $k = 11$

Figure 2.11: Encoding performance of the STAR code for $k = 11,\ w = 10$, and $m = 3$.

Code	k	w	m
EVENODD	11	10	2
RDP	10	10	2
X-code	11	13	2
STAR	11	10	3

Table 2.8: Encoding parameters of various codes.

The results are in Figure 2.8 to Figure 2.11. In terms of peak performance, the codes match expectations. When $k = w$, the Liberation codes' performance is theoretically identical to EVENODD [90], and this is reflected in the results. RDP and X-code should encode the fastest, and they do. The STAR code's performance is worse than the others because it encodes an extra parity device.

In terms of the trends, the codes' performance matches the Liberation codes, and all the observations for the Liberation codes hold for these codes as well. The results show that the performance comparison of the scheduling algorithms is consistent for a wide variety of codes.

2.5.5 Encoding Performance of RDP

In [31], Corbett et al. provide encoding performance of RDP code, and it is the best reported result of RDP in the literature. The machine used in [31] contains two Pentium 4 CPUs of 2.8GHz, and one CPU is dedicated for RDP encoding. They do not report the machine word size and the cache sizes of the CPU. The most similar machine in our tests is machine P4, but P4 only has one CPU. The performance comparison of *DWG* XOR-scheduling with their reported performance is given in Table 2.9.

Machine	DWG XOR-scheduling	Reported in [31]
P4	2.186 GB/s	1.55 GB/s

Table 2.9: Encoding performance of RDP for $k = w = 10$ and $m = 2$.

Table 2.9 shows that the encoding performance achieved by *DWG* XOR-scheduling is much higher than their reported performance. Without knowing their exact implementation details, the reasons can be many-fold. However, a simple conclusion to draw is that *DWG* XOR-scheduling, an algorithm proposed in this chapter, can achieve much higher performance than existing reported results.

2.6 Summary

This chapter studies efficient XOR-scheduling algorithms to improve the encoding performance of XOR-based erasure codes. For these erasure codes, the encoding performance is determined by two primary

factors: the number of XOR operations and the cache behavior. Two new proposed algorithms, named Data Words Guided XOR-scheduling and Data Packets Guided XOR-scheduling, are able to efficiently utilize CPU cache and thus achieve much better encoding performance than the traditional algorithm. In a performance evaluation on some widely known erasure codes on a variety of machines, we show that the encoding performance obtained by Data Words Guided XOR-scheduling considerably outperforms that of the traditional algorithm; although Data Packets Guided XOR-scheduling improves the performance less significantly than Data Words Guided XOR-scheduling, it has more stable performance across various packet sizes on different machines.

CHAPTER 3 Efficient Erasure Decoding for RAID-6 Codes

3.1 Introduction

Reliable storage systems are essential for most data related applications. Yet virtually every component in a storage system can fail, from hard disk drives to various cables. Most recent studies show hard disks, the core component of storage systems, fail much more often in real systems than specified in their data-sheets [86, 98]. Inevitably, data redundancy needs to be introduced to ensure intact recovery of lost data caused by disk failures. A quite common practice is to use mirroring by maintaining multiple copies of a data on different disks or storage nodes. As storage systems expand and use more and more commodity components, it is common nowadays to employ *triplication* or more in storage systems, such as the Google File System (GFS) [40]. It is arguable, however, if such a practice uses system resources in a most effective and efficient way. GFS uses not just the triple number of disks, but also consumes other expensive resources, including network components, power and physical spaces, let alone a great deal of man power to manage and maintain the system.

On the other hand, RAID-5 [85], another commonly adopted data redundancy architecture that uses one parity disk/node to tolerate one disk/node failure, may not have high enough reliability in real systems. Schroeder et al. observed that disks fail in a more *correlated* fashion [98], and thus theoretical reliability of RAID-5 derived on the assumption that disks fail *independently* may be very far from reality. A natural extension to provide higher data reliability is to use more redundancy, and RAID-6 which employs *two* parity disks/nodes is a good starting point. Here RAID-6 is used as a general data redundancy architecture, not necessary limited to *disk arrays* with a RAID controller: the same data placement schemes can be certainly extended to *clustered* or *distributed* storage systems. In fact, this is true for any RAID-n architecture. For discussion simplicity, the term RAID will be used throughout this chapter, but by no means any results hereafter are limited to disk arrays.

One view of disk failures for RAID-6 is *stop-and-fail* error model. Under this model, a disk either functions normally - all data stored on it is accessible; or fails totally - none of its stored data is accessible,

which is called *complete disk failure*. In a practical storage system, disk failures are certainly more complex. In general, some sectors or blocks of a disk can fail or corrupt because of various reasons [13]. This is the so-called *sector failures*. Prabhakaran et al. gives a discussion on the common factors resulting in sector failures [95]. Very recent study shows that actual sector failures occur much more frequently than an complete disk failure in real storage systems [13], and sector failures become a significant threat to data reliability [14]. As modern disks exponentially increase their storage capacity by increasing their areal density, it is anticipated that sector failures will occur more often. To avoid the data lost due to sector failures, Schwarz et al. proposed a technique called disk *scrubbing* [101], which runs a background process and proactively scans disk sectors masking failed ones.

Now let's consider the complexity of disk failures in RAID-6 storage systems. As a RAID-6 system contains several disks, and each disk may encounter entire disk failure or sector failures, disk failures in the whole system could become complicated. Here, we only focus on recoverable disk failures, for which lost data can be completely reconstructed from surviving disks. We distinguish four different types of recoverable disk failures in RAID-6 systems, and they are: 1) sector failures (sector failures could spread out on multiple disks), 2) one complete disk failure, 3) one complete disk failure with sector failures on other disks, 4) two complete disk failures. Then, to protect data from disk failures, it is crucial for a RAID-6 system to use a decoding algorithm that can reconstruct data from all the four disk failure types. Moreover, the decoding algorithm should perform in an efficient manner, for two purposes. First, during data reconstruction process, the whole system may not allow data access, and hence the faster data reconstruction, the shorter downtime of the system. Second, when disk failure happens, a storage system's reliability is more subject to be compromised than normal because it is in the window of vulnerability (WOV) [14]. As a result, efficient decoding can help to reduce WOV and improve storage systems' reliability.

In this chapter, we study how to efficiently decode disk failures for RAID-6 codes. The contributions of the chapter are both theoretical and practical. In theory, we present the sufficient and necessary conditions to determine if a disk failure is recoverable or not. The conditions are valid for many well known codes, such as X-code [115], EVENODD [20], RDP [31], and WEAVER codes [47]. Practically, a *universal* decoding algorithm named *SCAN* is designed. The *SCAN* algorithm can efficiently correct all theoretically recoverable disk failures. An implementation of the *SCAN* algorithm is also provided to cover all different types of disk failures. In addition, we evaluated the performance of the *SCAN* algorithm for X-code, EVENODD, and RDP. The performance results show the *SCAN* algorithm significantly outperforms an existing approach called *Matrix Method* [49].

3.2 Related Work

In this section, we introduce related coding theory and decoding algorithms for RAID-6 codes.

3.2.1 Coding Theory

An (n, k) erasure code uses mathematical means to transform a user data of k data symbols into a block of n (same size) symbols by adding $(n - k)$ parity symbols. Each parity symbol is computed using a parity constraint (usually a linear equation) from the k data symbols. The resulting n-symbol block is called a *codeword*. This computation process of obtaining a codeword from k data symbols is called *encoding*. Similarly, the process of retrieving k data symbols from a codeword is called *decoding*. In a codeword, symbol is a logical unit; it can be a bit, a byte, or multiple bytes. Previous work has shown that large symbol size is likely to achieve high encoding/decoding performance [77, 92].

With respect to encoding/decoding, erasure codes have two basic matrices: generator matrix and parity check matrix [79]. A generator matrix defines how parity symbols are computed from data symbols. A parity check matrix is derived from generator matrix, and it can verify the parity constraints of data and parity symbols in a codeword. If a codeword is treated as a symbol vector c and parity check matrix as matrix P, then we have $c \cdot P = 0$ if the codeword has no error. In this chapter, we focus on XOR-based RAID-6 erasure codes, whose encoding/decoding contains only binary exclusive-or computation, and hence the element value in their parity check matrices is 0 or 1.

When using coding technology to build RAID storage systems, user data received from applications is encoded to form codewords, and the codewords are striped to multiple disks. For an (n, k) code, each codeword contains n symbols and they are distributed on different disks. When disk failure happens, the symbols stored on the failed disks or failed sectors are all lost, and these lost symbols have to be reconstructed from surviving disks. If there are k surviving disks, the lost symbols can be recovered and no data is lost. The lost symbols are called *erasures*, and the reconstruction process is an erasure decoding operation.

3.2.2 Decoding Algorithms for Complete Disk Failures

When introducing a new RAID-6 code, the code inventors often provide a decoding algorithm for it. This is the case of X-code [115], EVENODD [20], and RDP [31]. In general, these decoding algorithms are designed to reconstruct data for only complete disk failures. Usually, they are not compatible with each other. This is because the decoding algorithm designed for one code is optimized and aims for best decoding

performance for that particular code. This type of algorithms is referred to as *decoding algorithms for entire disk failures* in this chapter.

The decoding algorithms for entire disk failures, however, commonly suffer from two limitations. First, they can not efficiently deal with sector failures. When encountering sector failures, this type of algorithms has to treat the disks containing the failed sectors as entire failed disks and then decode. Hence, besides real erasures, this approach also recover those normal sectors which fall on the assumed failed disks, resulting in unnecessary computation. Second, they may fail to reconstruct data for a disk failure event even when it is recoverable. For example, if a failure event contains only sector failures but they are spread over three disks, this type of decoding algorithms will assume that there are three entire failed disks and declare it not recoverable. However, this kind of failure events may be still recoverable. We will show such an example in Section 3.4.2. Therefore, the above two shortcomings limit the usage of this type of decoding algorithms in practical RAID-6 systems considering sector failures.

3.2.3 Decoding Algorithms for Sector Failures

This section describes general decoding algorithms that can recover any type of disk failures. These algorithms treat complete disk failures as a special case of sector failures, and then they use same approach to deal with all disk failure types.

Iterative Decoding

RAID-6 codes can be represented as Low-Density Parity-Check (LDPC) codes [37] at symbol level. LDPC codes are erasure codes that are constructed from sparse parity check matrix, and they can be decoded by using *Tanner graph* [105, 91]. A Tanner graph is a bipartite graph. The nodes in the graph are partitioned to two disjointed groups. One group contains erasure nodes denoting erasures, and the other group only contains parity constraint nodes. Then, the two groups are connected according to their parity constraints, which are derived from the code's parity check matrix. From linear algebra perspective, each erasure node stands for a variable and each parity constraint node represents a linear equation, and decoding operation is essentially a linear equation solving process.

Iterative decoding is a simple decoding algorithm using Tanner graph. It works as follows. It first builds a Tanner graph for a codeword with erasures. Then, it finds a constraint node of degree 1, uses this constraint node to recover its connected erasure node, and removes the two nodes from the graph. This step is repeated until no constraint node of degree 1 is left. At the end, if there are no erasures left, all erasure are recovered;

otherwise, some erasures are not recoverable. The main drawback of this algorithm is that it may not be able to decode a codeword even when it is recoverable. To the best of our knowledge, no previous work has discussed this problem before, i.e., what kind of codewords can be decoded by iterative decoding. In this chapter, we partially address this issue. We identify a family of RAID-6 codes that can be completely decoded by iterative decoding.

Matrix Method

Hafner et al. proposed a general decoding approach, called *Matrix Method* [49]. When decoding a codeword, the *Matrix Method* first builds a matrix called *workspace matrix* based on the code's parity check matrix. Then, it processes the codeword's erasures one by one. For each erasure, it first removes the erasure from the workspace matrix and then transforms the matrix so that the processed erasures can be reconstructed from surviving symbols. After all erasures are processed, the workspace matrix turns out to be a *recovery matrix*, which instructs how the erasures can be reconstructed. Based on the above basic algorithm, Hafner et al. also provided several optimization techniques to improve time and space efficiency for the *Matrix Method* [28, 49].

The *Matrix Method* is a general decoding algorithm for XOR-based erasure codes; however, it has the following limitations: 1) deciding when to properly use the optimization techniques proposed for the *Matrix Method* is a challenge. These techniques presented in [49, 28] are not stated clearly when they should be applied, and their inappropriate use may even worsen the performance, 2) because *Matrix Method* is a general decoding approach and not designed specially for RAID-6 codes, the *Matrix Method* can not work efficiently for all possible disk failures even with its various optimization techniques, which will be shown in Section 3.6.

3.3 *ED-2* Codes

Now we introduce a family of RAID-6 codes, which is called *ED-2* codes.

3.3.1 Lemma

An *erasure pattern* identifies the set of erasures in a codeword. Then, we have the following definition for the recoverability of an erasure pattern.

Definition 1. *If all erasures in an erasure pattern can be uniquely recovered, then this erasure pattern is recoverable.*

If the erasures in erasure pattern A are all contained in erasure pattern B, then B is called a *superset* of A. It is easy to see that an erasure pattern and any of its supersets have the following relation:

Lemma 1. *If an erasure pattern is not recoverable, then any of its supersets is not recoverable.*

Lemma 1 implies that an erasure pattern is irrecoverable if it contains an irrecoverable subset. Though simple, this lemma gives a necessary condition for a recoverable erasure pattern.

For the rest of this chapter, we focus on RAID-6 codes, a type of erasure codes capable of tolerating any two complete disk failures. If a code can tolerate the failure of more than two disks, we simply uses its derived code with the capability for two disks.

3.3.2 *ED-2* Codes

Definition 2. *Given a code, for an arbitrary erasure pattern, if every erasure node in its Tanner graph has degree of at most 2, then this code is an* ED-2 *(Erasure Degree 2) code.*

First, we give a sufficient condition to determine if a code is an *ED-2* code. For a code, if the number of 1s in any row of its parity check matrix is no more than 2, then this code is an *ED-2* code. Recall that in parity check matrix, one column represents a parity constraint, and one row contains the parity constraints one symbol is involved. Then, if the number of 1s in any row is no more than 2, for any erasure pattern, any erasure node has at most degree of 2 in the Tanner graph. Therefore, this code must be an *ED-2* code. From this sufficient condition, we find several *ED-2* codes and list them in Table 3.1. It is worthy noting that an *ED-2* code does not have to be a *maximum distance separable* (MDS) code.

RAID-6 Code	*ED-2* Code	*MDS*
B-code	YES	YES
X-code	YES	YES
BCP code [15]	YES	YES
ZZS code [119]	YES	YES
WEAVER codes	YES	NO
LSI code [112]	YES	NO
EVENODD	NO	YES
RDP	NO	YES

Table 3.1: *ED-2* codes

Example 1: X-code [115] is a MDS RAID-6 code, and it has optimal encoding/decoding/update efficiency. A codeword of X-code can be represented by an $n \times n$ array, where n is a prime number. For $n = 5$,

the construction of X-code is shown in Figure 3.1. In Figure 3.1, the top three rows contain data symbols, and the bottom two rows contain parity symbols. A parity symbol with a shape is the bitwise XOR sum of data symbols with the same shape.

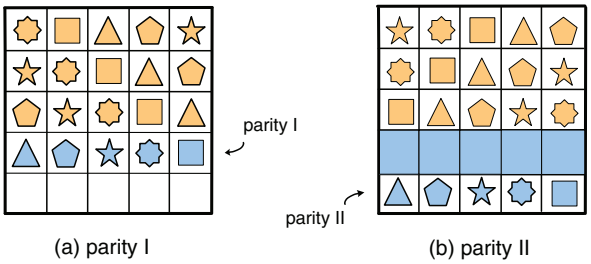


Figure 3.1: X-Code (5, 3) construction

Table 3.1 shows that X-code is an *ED-2* code. Now, we use X-code as an example to describe which erasure patterns of *ED-2* codes are recoverable and which are not.

3.3.3 Recovery Theorem

Theorem 1. *For an* ED-2 *code, if an erasure pattern is* ***irrecoverable****, then its Tanner graph must have one of the following subgraphs:*

1. *A cycle;*
2. *A path staring from an erasure node of degree 1 and ending at another erasure node of degree 1.*

Proof. First the *sufficient* condition part: if an erasure pattern has any subgraph given in this theorem, this pattern is irrecoverable. If we can prove that the erasure pattern characterized by either subgraph is not recoverable, then we can show that the original erasure pattern is not recoverable according to Lemma 1. Below we consider the two subgraphs one by one.

1. A cycle: it implies that the number of erasure nodes is the same as that of constraint nodes. As a result, the equations represented by the constraint nodes are linearly *dependent*, and thus the solutions to the erasures in the cycle are not unique. Hence, the erasures in the subgraph are irrecoverable by Definition 1.

   Figure 3.2 shows such an example. In this example, a cross sign (X) denotes an erasure. Erasure node $e_{i,j}$ represents an erasure at row i and column j, and constraint node $E_{i,j}$ represents the parity constraint imposed by the parity symbol at row i and column j. These nodes are connected according to the parity check matrix of X-code.

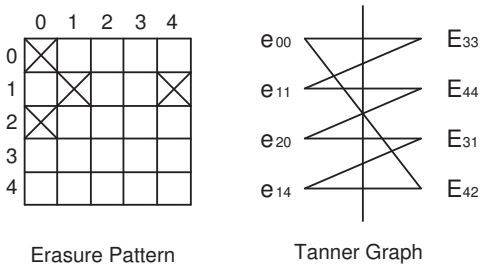


Figure 3.2: An erasure pattern of X-code.

2. A path staring from an erasure node of degree 1 and ending at another erasure node of degree 1: as a Tanner graph is a bipartite graph, then in this path, the number of erasure node is one greater than the number of constraint nodes. From equation solving perspective, it implies that the number of variables is one greater than the equation number, and hence the solutions for the erasures are not unique. Therefore, the erasures in this subgraph are also irrecoverable by Definition 1. Figure 3.3 shows such an example.

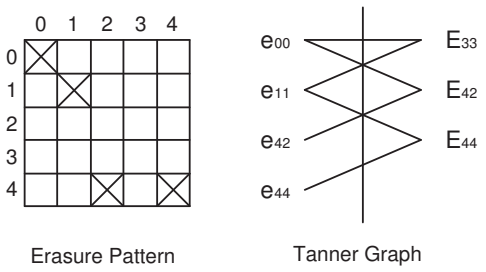


Figure 3.3: An erasure pattern of X-code.

Now the *necessary* condition part: if an erasure pattern is irrecoverable, it must contain one subgraph listed in the theorem. Assume there is an irrecoverable erasure pattern of an *ED-2* code. Then, we use iterative decoding [105] to decode it. When iterative decoding can not proceed, we randomly pick up a connected component to observe. We can find this component must belong to one of the following two cases:

1. it contains a cycle: it is the first subgraph.

2. it does not contain a cycle: then there must be a path staring from a node of degree 1 and ending at another node of degree 1; otherwise there exists a cycle. Either of these two nodes can not be constraint nodes because iterative decoding has processed all constraint nodes of degree 1. Hence, these two nodes must be erasure nodes, and this path is exactly the second subgraph.

□

In fact, the above proof of the necessary condition part infers the following corollary.

Corollary 1. *For an* ED-2 *code, iterative decoding is able to recover any recoverable erasure pattern.*

Furthermore, from Theorem 1, we can deduce the following theorem concerning recoverable erasure patterns:

Theorem 2. *For an* ED-2 *code, if an erasure pattern is* ***recoverable****, then its Tanner graph does not contain any of the following subgraphs:*

1. *A cycle;*

2. *A path staring from an erasure node of degree 1 and ending at another erasure node of degree 1.*

3.3.4 *SCAN* Algorithm

Now we turn to an erasure decoding algorithm, called *SCAN* algorithm. The *SCAN* algorithm is essentially an iterative decoding algorithm, so it can decode *ED-2* codes. However, it differs from general iterative decoding in two aspects. First, the *SCAN* algorithm adopts the Zig-Zag approach [20, 57]. For an *ED-2* code, as any erasure node in Tanner graph has no more than degree 2, the Zig-Zag approach is highly efficient in performing iterative process. Second, the *SCAN* algorithm is not only applicable to *ED-2* codes but also to two other codes, EVENODD and RDP, which can not be decoded by general iterative decoding. This will be shown later.

A pseudocode of the *SCAN* algorithm is shown in Algorithm 5. The *SCAN* algorithm consists of two operations: *Check* operation and *Recover* operation. Loosely speaking, the *Check* operation builds a recovery plan for an erasure pattern, and then the *Recover* operation proceeds to correct the erasure pattern if it is recoverable. If an erasure pattern is partially recoverable, i.e., not all the erasures can be recovered, the *Recover* operation will be skipped. (Note: The *SCAN* algorithm is able to recover all recoverable erasures for an irrecoverable erasure pattern. The current choice is made in order to be consistent with our definition of *recoverable* erasure pattern.) In Algorithm 5, the *Check* operation collects the list of erasures in *ERASURE_LIST*, and the *Recover* operation recovers the erasures in the list one by one.

3.4 *SCAN* for EVENODD

This section presents how to determine if an erasure pattern is recoverable for EVENODD, and it provides a variant of *SCAN* algorithm for the code.

Algorithm 5 The *SCAN* Algorithm

procedure $SCAN()$

1: Check();
2: Recover();

procedure $Check()$

1: Build Tanner graph;
2: **for** each constraint node CN **do**
3: **if** the degree of CN is 1 **then**
4: ZigZag(CN);
5: **end if**
6: **end for**

procedure $ZigZag(CN)$

1: Get CN's connected erasure node eN;
2: **if** eN does not exist **then**
3: return;
4: **end if**
5: Add eN to *ERASURE_LIST* list;
6: Find the other constraint node *CN2* connecting to eN;
7: Remove CN and eN from the graph;
8: **if** $CN2$ exists and its degree is 1 **then**
9: ZigZag(*CN2*);
10: **end if**

procedure $Recover()$

1: Recover eN in *ERASURE_LIST* one by one;

3.4.1 EVENODD Description

EVENODD [20] is an MDS RAID-6 code. It uses 2 parity columns together with p data columns, where p is a prime number. In EVENODD, the 1st parity column is in column p, which is computed as the XOR sum of all data symbols in the same row. This parity column is called *horizontal* parity column. The 2nd parity column is in column $(p+1)$, which takes the following steps to compute: first, the XOR sum of all data symbols along the same diagonal (indeed a diagonal of *slope* 1) is computed and assigned to their corresponding parity symbols in column $(p+1)$ except the main diagonal value called the *adjuster*; second, the *adjuster* is added (XOR addition) to all parity symbols in column $(p+1)$, which is *adjuster complement*. We then call parity column $(p+1)$ *diagonal* parity column. For $p = 5$, the construction of EVENODD is shown in Figure 3.4.

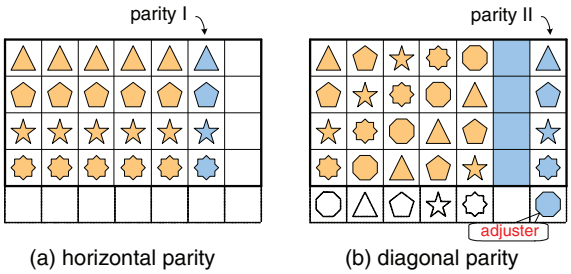


Figure 3.4: EVENODD (7, 5) construction.

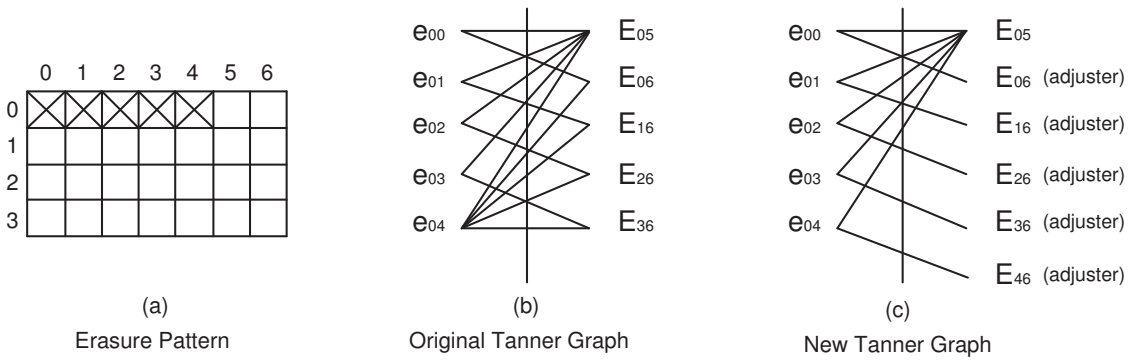


Figure 3.5: An example of new Tanner graph for EVENODD.

3.4.2 Tanner Graph of EVENODD

We first show EVENODD is not an *ED-2* code through an example. Figure 3.5(a) is an erasure pattern of EVENODD, and Figure 3.5(b) is its Tanner graph. In this Tanner graph, erasure node e_{04} has a degree of 5, which is more than 2, and thus EVENODD does satisfy to the definition of *ED-2* codes. Here, the distinction

of erasure node e_{04} from other erasure codes is that it is an erasure positioned at the main diagonal. Due to the computation of *adjuster complement*, the erasure nodes representing the erasures in the main diagonal need to connect to all diagonal parity constraint nodes. Hence, in this case, the degree of e_{04} is 5. Note that although this erasure pattern is recoverable, it can not be decoded by iterative decoding because there is no parity constraint node of degree 1 to start with.

Here, we present an approach to modify the construction of the Tanner graph so that EVENODD can become an *ED-2* code. In the Tanner graph of EVENODD, we classify the constraint nodes to two types: *horizontal* constraint nodes and *diagonal* constraint nodes. They represent the parity constraints imposed by horizontal parity symbols and diagonal parity symbols respectively. Then, we construct the new Tanner graph as follows:

1. Add a new diagonal constraint node to represent the constraint imposed by the main diagonal;
2. If a data symbol is an erasure, its erasure node only connects to its horizontal constraint node and diagonal constraint node.

The new Tanner graph for the erasure pattern of Figure 3.5(a) is shown in Figure 3.5(c). Figure 3.5(c) differs from Figure 3.5(b) in: 1) a new constraint node E_{46} is added, 2) four edges in Figure 3.5(b) are removed, 3) a new edge (e_{04}, E_{46}) is added. Now, e_{04} connects only to E_{05} and E_{46}, and its degree reduces to 2. It is worth noting that in the figure, the *adjuster* is placed at the right side of all diagonal constraint nodes. This is because now the *adjuster* is an implicit variable and it occurs at all equations represented by diagonal parity constraint nodes. For example, in Figure 3.5(c), we have $e_{00} = adjuster$, and $e_{01} = adjuster$ etc.

From equation solving perspective, the new Tanner graph is equivalent to the original Tanner graph, but the new graph is much simpler and easier to process. Next section will show how to compute the *adjuster* from the new graph. After the *adjuster* is computed, EVENODD becomes an *ED-2* code, and all previous results of *ED-2* codes can be applied to it. Actually, the decoding with the new Tanner graph shares the similar way with encoding process. In encoding process, the *adjuster* is first computed to improve encoding performance; in this decoding approach, the *adjuster* is first recovered for better decoding performance.

3.4.3 Recovery Theorem of EVENODD

Recovery of the *adjuster*

Theorem 3. *For an arbitrary erasure pattern, if the **adjuster** is recoverable, its new Tanner graph must have a connected component satisfying the following conditions:*

1. *the degree of each erasure node is 2;*

2. *the number of diagonal constraint nodes is odd.*

Proof. First the *sufficient* condition part: if a connected component satisfies the two conditions, the *adjuster* can be recovered. When we have such a connected component, we add (XOR) all the equations represented by this component and get a single equation. We can find that this equation has two properties: 1) at the left side, there is no variable left, because the degree of each erasure node is 2 and their corresponding variables are all canceled by XOR operation; 2) at the right side, there is only one variable left and it is the *adjuster* due to the number of diagonal constraint nodes being odd. As a result, the *adjuster* can be computed from this equation.

Now the *necessary* condition part: if no component satisfies the two conditions, the *adjuster* is not recoverable. As such, any component must be one of the following cases:

1. The first condition is not satisfied. It means this component contains some erasure nodes of degree 2 and some others of degree 1. Suppose there are totally m erasure nodes in this component. Then, the number of constraint nodes is at most m. Therefore, this component contains $m + 1$ variables represented by m erasure nodes plus the implicit variable (the *adjuster*) but there are at most m equations, and thus the solutions are not unique for these variables. As a result, the *adjuster* is not recoverable.

2. The first condition holds, but the second condition is not satisfied. It implies the degree of each erasure node is 2 but the number of diagonal constraint nodes is even. Suppose there are m erasure nodes in this component. Then, there are at most $m+1$ constraint nodes in the component. Now we have at most $m + 1$ equations, and we can add (XOR) all these equations to get one equation. In the final equation, all variables are canceled since each variable occurs twice, so there are actually at most m independent equations. Thus, the component has $m + 1$ variables but there are at most m independent equations. Again, the solutions to the equations are not unique, and hence the *adjuster* is not recoverable.

□

Recovery Theorem

Lemma 2. *A* necessary *condition for a recoverable erasure pattern of EVENODD is that the* adjuster *is recoverable.*

Proof. Recall that the *adjuster* is the XOR sum of all the data symbols along the main diagonal, so when an erasure pattern is recoverable, all the data symbols along the main diagonal are recoverable, and the *adjuster* is certainly recoverable then. □

By combining Theorem 2 and 3, and Lemma 2, we can get the following theorem for EVENODD.

Theorem 4. *For EVENODD, if an erasure pattern is recoverable, its **new** Tanner graph must satisfy the following conditions:*

1. *one connected component satisfies the conditions given in Theorem 3;*
2. *all connected components satisfy the conditions given in Theorem 2.*

3.4.4 *SCAN* Algorithm for EVENODD

The decoding algorithm for EVENODD is a variant of the *SCAN* algorithm for *ED-2* codes. The variant contains the following two changes:

1. In the *Check* operation: a new Tanner graph is built, and we need to determine whether the *adjuster* is computable according to Theorem 3;
2. In the *Recover* operation: the *adjuster* should be recovered first before any diagonal constraint node is used to recover an erasure node.

When implementing the *Check* operation, determining whether the *adjuster* is computable can be piggy-backed in the $ZigZag$ process. This makes the *Check* operation for EVENODD have similar time complexity to that for ordinary *ED-2* codes. In the *Recover* operation, when a parity constraint node is used, the XOR number to recover one erasure is exactly $(p-1)$. As the cost of computing the *adjuster* can be amortized to all the erasures, it is not significant though. Since for a general $(p+2, p)$ MDS code, the lower bound of XOR number needed to recover one erasure is $(p-1)$, the time complexity in the *Recover* operation of SCAN algorithm is close to the lower bound.

3.5 *SCAN* for RDP

Like EVENODD, RDP is not an *ED-2* code. In this section, we present a new way to construct the Tanner graph for RDP. Based on the new graph, we develop a variant of the *SCAN* algorithm for RDP.

3.5.1 RDP Description

RDP [31] is a $(p+1, p-1)$ MDS code, where p is a prime number. RDP has two parity columns. One parity column is called *horizontal* parity column which stores horizontal parity symbols, and another parity column is called *diagonal* parity column storing diagonal parity symbols. The diagonal parity column in RDP is close to that in EVENODD. The difference is that RDP uses horizontal parity symbols to compute diagonal parity symbols. For $p = 5$, the construction of RDP is shown in Figure 3.6. Again, a parity symbol with a shape is the XOR sum of other symbols with the same shape.

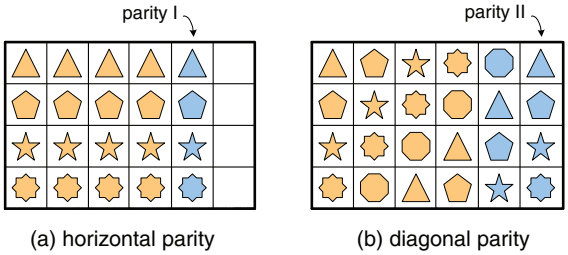


Figure 3.6: RDP (6, 4) Construction

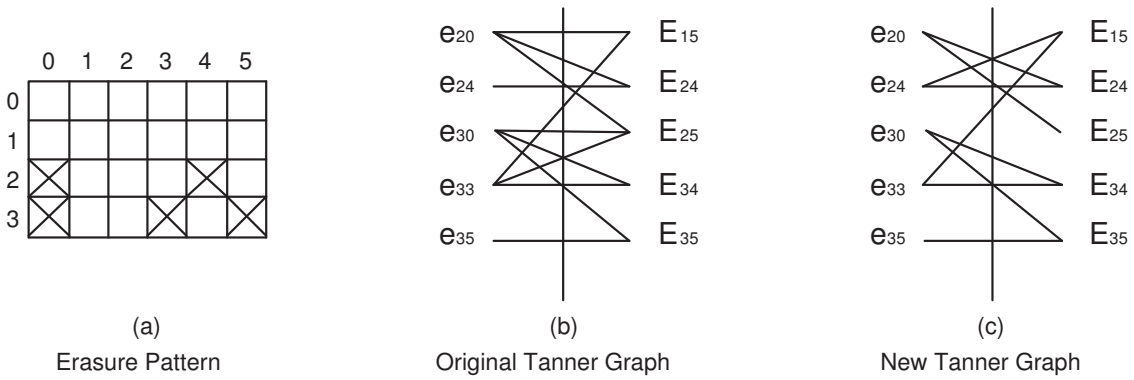


Figure 3.7: An example of new Tanner graph for RDP.

3.5.2 Tanner Graph of RDP

RDP is not an *ED-2* code, which can be shown by an example in Figure 3.7. Figure 3.7(a) is an erasure pattern, and Figure 3.7(b) is its Tanner graph. In the graph, erasure nodes e_{20}, e_{30}, and e_{33} all have degree 3, demonstrating that RDP is not an *ED-2* code.

Similar to EVENODD, the original Tanner graph of RDP can be modified so that RDP can become an *ED-2* code. First, we define two terms. In Figure 3.7(b), E_{25} is defined as the $direct$ diagonal constraint node of e_{20} because parity symbol p_{25} is computed from data symbol d_{20} during encoding process. E_{15} is

the $associated$ diagonal constraint node of e_{24} because parity symbol p_{25} is computed from parity symbol p_{24} in encoding. Now, the tanner graph can be constructed as follows:

1. If a data symbol is an erasure, its erasure node only connects to its horizontal constraint node and $direct$ diagonal constraint node;

2. If a horizontal parity symbol is an erasure, its erasure node connects to its horizontal constraint node and its $associated$ diagonal constraint node.

Figure 3.7(c) is the new Tanner graph. From equation solving perspective, the new Tanner graph is equivalent to the original Tanner graph but much simpler. Compared to Figure 3.7(b), Figure 3.7(c) removes three edges and adds a new edge (e_{24}, E_{15}). Now the degree of all erasure nodes is no more than 2, and the new Tanner graph satisfies the conditions of *ED-2* codes. It can be seen that the construction of new Tanner graph of RDP shares the same manner with its encoding process on using horizontal parity symbols. It suggests that observing a code's encoding process may be helpful to design an efficient decoding algorithm.

3.5.3 Recovery Theorem of RDP

Different from EVENODD, the new Tanner graph of RDP has no implicit variable, and then its recovery theorem (shown in Theorem 5) is simpler than that of EVENODD. The proof of this theorem is almost the same as that for *ED-2* codes (presented in Theorem 2), so it is skipped here.

Theorem 5. *If an erasure pattern of RDP is recoverable, its* ***new*** *Tanner graph does not have any of the following subgraphs:*

1. A cycle;

2. A path staring from an erasure node of degree 1 and ending at another erasure node of degree 1.

3.5.4 SCAN Algorithm for RDP

The *SCAN* algorithm for RDP is almost the same as that for *ED-2* codes, and the only change is in the *Check* operation, a new Tanner graph has to be built. Regarding to the complexity of the *SCAN* algorithm, the *Check* operation needs a small factor of p to generate a recovery plan for a $(p+1,\ p-1)$ RDP, and the *Recover* operation uses only $(p-2)$ XOR operations to recover each erasure. This number is the lower bound for a $(p+1,\ p-1)$ RDP.

3.6 Performance Evaluation

Now we evaluate the decoding performance of the *SCAN* algorithm for three codes: X-code, EVENODD, and RDP.

3.6.1 Experiment Setup

Evaluated Algorithms

Matrix Method [49] is another general decoding algorithm for RAID-6 codes. This algorithm has a basic algorithm and several optimization techniques. The basic algorithm, called *Column-Incremental Construction*, generates a recovery matrix for an erasure pattern. As the number of 1s in the recovery matrix determines the number of XOR operations, several techniques are proposed to reduce this number [49]. One technique is called *Reversing The Column Incremental Construction*. Simply put, after an erasure is recovered, this technique treats the recovered erasure as a normal symbol and updates the recovery matrix accordingly, and then this symbol can be used to recover the left erasures. Like the *SCAN* algorithm, the *Matrix Method* also contains two steps, *Check* operation and *Recover* operation.

We implement both the *SCAN* algorithm and the *Matrix Method*. For the *Matrix Method*, we implement its basic algorithm (*Column-Incremental Construction*) and call it *Basic Matrix Method*. We also implement the basic algorithm with the optimization technique (*Reversing The Column Incremental Construction*) and call it *Matrix Method (Reverse)*. All implementations are in C language and compiled by *gcc* with $-O2$ optimization flag, recommended for most applications running in real environments [39].

Experiment Environment

We run experiments for three codes: X-code representing *ED-2* codes, EVENODD, and RDP. All test data is loaded into main memory, so no real file I/O is involved. For each code, several typical pairs of (n, k) are tested. In all pairs, we have $n = k + 2$. Symbol size is fixed to be 512 bytes in all codewords, and hence one symbol corresponds to one disk sector.

All experiments are conducted on a HP dc7600 workstation with following relevant hardware and software configurations: a) CPU: Intel Pentium D 2.8GHz; b) Memory: DDR2 533MHz, 1GB; and c) Kernel: Linux 2.6.18.2-34-default X86-64. Each experiment is repeated 30 times to measure the average value of decoding performance. The test results show the margin of error to the average value is no more than 5%, and thus error bars are skipped in the following figures.

3.6.2 Sector Failures

Failure Model

We first show the results for sector failures. Here, we use the Gilbert model [42] to characterize sector failures. The Gilbert model is a common finite-state Markov model for *bursty* failures, as depicted in Figure 3.8.

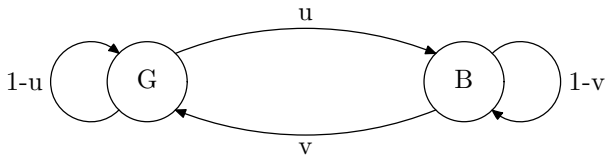
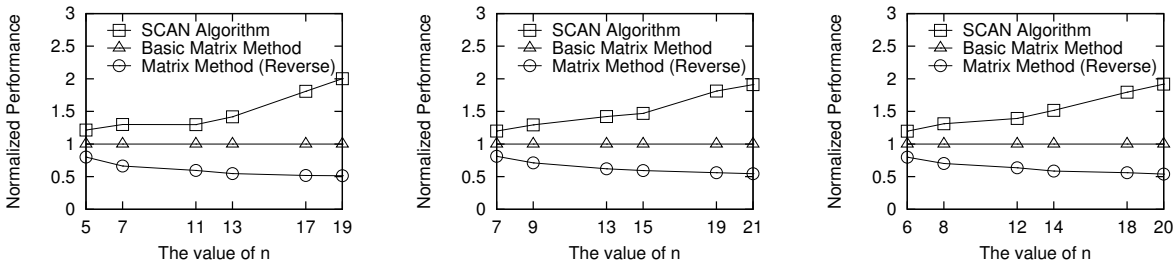


Figure 3.8: The Gilbert Model

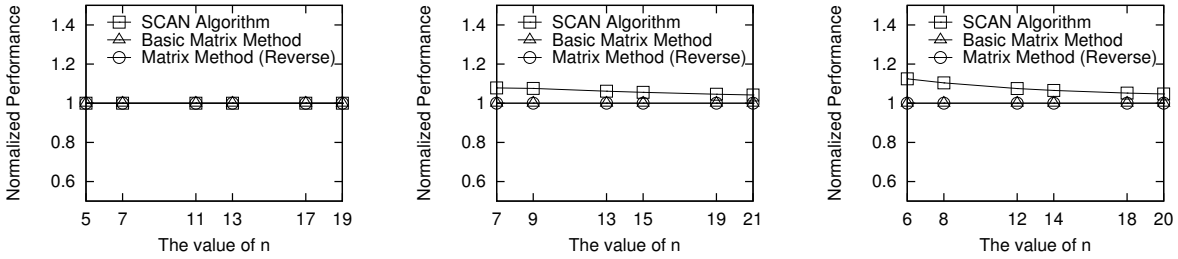
For a hard disk, the same factors causing one sector failure may very likely affect a few neighboring sectors and thus cause those sectors to fail as well, as observed by L. Bairavasundaram et al. [13]. Hence, the Gilbert model closely captures the characteristics of sector failures. Using this model, a disk sector is either in state G (good or normal) or state B (bad or failure). Assume s_1 and s_2 are two consecutive sectors and s_2 follows s_1. If s_1 is in state G, then s_2 is in state B with a probability of u and in state G with a probability of $1 - u$. State B has the transition probability of v. The *stationary failure rate* - the overall sector failure rate or probability - is $u/(u + v)$ when the disk failure becomes stable. For a group of disks, we assume each disk *independently* follows the Gilbert model but with the same (u, v) [98]. This is a simple but yet reasonable model.

Based on previous research on sector failure probability [13], we choose 10^{-9} for the value of u and 0.5 for v. Then, we generate totally $100,000$ erasure patterns to decode. Each one contains at least 1 erasure, and all of them are recoverable.



(a) X-code (b) EVENODD (c) RDP

Figure 3.9: Decoding performance comparison for sector failures



(a) X-code (b) EVENODD (c) RDP

Figure 3.10: Decoding performance comparison for one complete disk failure

Performance Comparison

For sector failures, we measure the total time consumed in decoding. Here, we don't use the absolute decoding time to compare the performance of decoding algorithms; instead, we present the comparison of normalized performance, calculated as below equation:

$$\text{Norm. Perf.} = \frac{\text{Decoding Time of } \textit{Basic Matrix Method}}{\text{Decoding Time of a decoding algorithm}}$$

Thus, the higher normalized performance is, the better decoding performance is achieved. The normalized performance of the decoding algorithms are presented in Figure 3.9. Following observations can be made from this figure:

1. For all n values, the *SCAN* algorithm performs better than *Basic Matrix Method*. The performance improvement of the *SCAN* algorithm over *Basic Matrix Method* ranges from 20% to 80%.

2. When n grows, the performance of the *SCAN* algorithm also increases. It implies that when the number of disks in RAID-6 system increases, the *SCAN* algorithm scales better than *Basic Matrix Method*.

3. In all cases, *Matrix Method (Reverse)* performs at least 20% worse than *Basic Matrix Method*.

For each algorithm, we can break down it to two operations – *Check* operation and *Recover* operation. We can find that the *Check* operation essentially dominates the decoding performance in this case. This is because when sector failure happens, there are only a few failed sectors due to the low probability of sector failures. Hence, in one codeword, there are only a few erasures, and the cost of the *Recover* operation is very small compared to the overall decoding cost. This is true for all decoding algorithms. It also explains why *Matrix Method (Reverse)* performs worse than *Basic Matrix Method*. Compared to *Basic Matrix Method*, *Matrix Method (Reverse)* has less efficient *Check* operation.

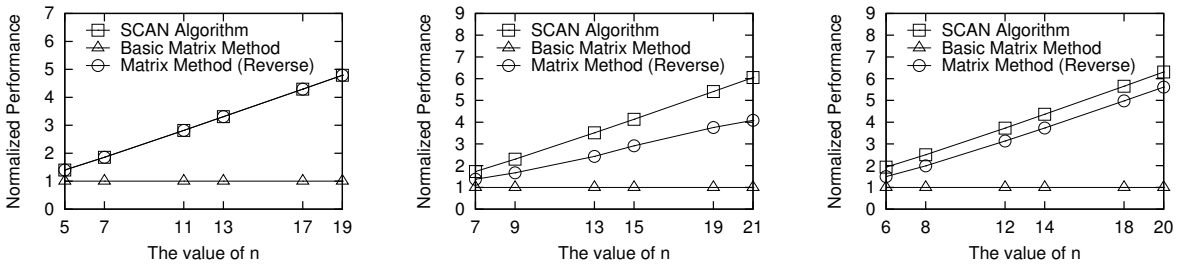
3.6.3 Complete Disk Failures

Different from sector failures, the decoding performance for complete disk failures is determined by the *Recover* operation rather than the *Check* operation. The reason is when complete disk failures happen, a great number of codewords share the same erasure patterns, and thus the *Check* operation needs to be performed only once, but the *Recover* operation has to be performed repeatedly for different codewords. Therefore, for this type of disk failures, the *Recover* operation dominates the decoding performance.

Since XOR is the main computation in the *Recover* operation, we use the number of XOR operations as the performance metric. We define normalized performance for complete disk failures as below:

$$\text{Norm. Perf.} = \frac{\text{XOR number of } \textit{Basic Matrix Method}}{\text{XOR number of a decoding algorithm}}$$

Again, the higher normalized performance is, the better decoding performance is achieved. We consider two cases for complete disk failures: one complete disk failure, and two complete disk failures. As more than two complete disk failures are beyond the recoverability of RAID-6 codes, they are ignored here.



(a) X-code (b) EVENODD (c) RDP

Figure 3.11: Decoding performance comparison for two complete disk failures

One Complete Disk Failure

When there is one entire disk failure, there could be sector failures in other disks. However, as sector failure is a low probability event, the decoding process will take most time to reconstruct data from one single disk failure without sector failures. Hence, here we only consider one pure complete disk failure.

To measure the decoding performance of one complete disk failure, we assume that each disk has the same failure probability, and thus the decoding performance is the average value of all possible single disk failures. The decoding performance is displayed in Figure 3.10, from which we can have the following observations:

1. For X-code, three decoding algorithms have the same decoding performance.

2. For EVENODD and RDP, *Basic Matrix Method* and *Matrix Method (Reverse)* achieve the same performance, but the *SCAN* algorithm obtains better performance and the performance improvement is about 5% to 15%.

Two Complete Disk Failures

When we consider two complete disk failures, we assume that the two disks are distributed to any two disks in a RAID-6 system with the same probability. Then, the decoding performance is the average value of all cases. Figure 3.11 displays the performance results, and we can observe:

1. For X-code, the *SCAN* algorithm performs 40% to 360% better than *Basic Matrix Method*. The performance improvement is even more for EVENODD and RDP.

2. The *SCAN* algorithm performs similarly to *Matrix Method (Reverse)* for X-code, but it obtains much better performance for EVENODD and RDP. The performance improvement range is 30-50% is for EVENODD, and it is 15% for RDP. The reason of why The *SCAN* algorithm outperforms *Matrix Method (Reverse)* is as follows. *Matrix Method (Reverse)* employs a technique called *Reversing The Column Incremental Construction* to determine if the erasure just processed should be used as intermediate results to recover the next unprocessed erasure. However, this is a greedy algorithm, and it can not always determine it correctly. But the *SCAN* algorithm is able to find all necessary intermediate results because when an erasure is processed, it automatically becomes an intermediate result and can be used to help recover unprocessed erasures. Therefore, the *SCAN* algorithm is more efficient than *Matrix Method (Reverse)*.

   As a matter of fact, for two complete disk failures, the *SCAN* algorithm is reduced to *decoding algorithms for entire disk failures*, because both algorithms need the same number of XOR operations in decoding. As the XOR number of *decoding algorithms for entire disk failures* is optimal, then that of the *SCAN* algorithm is also optimal.

3. *Matrix Method (Reverse)* obtains significantly better performance than *Basic Matrix Method* for all three codes. For instance, the performance improvement of *Matrix Method (Reverse)* over *Basic Matrix Method* ranges from 40% to 360% for X-code. It indicates the optimization technique utilized in *Matrix Method (Reverse)* is effective for this failure type.

In summary, for two complete disk failures, the *SCAN* algorithm improves the decoding performance over *Matrix Method (Reverse)* by up to 50% and over *Basic Matrix Method* by 40-500%. The reason of significant performance improvement is because the *SCAN* algorithm leverages the geometric structure of RAID-6 codes.

3.7 Summary

This chapter studies efficient decoding algorithms for RAID-6 codes, with contributions in both theory and practice. Theoretically, we introduce *ED-2* codes, a family of RAID-6 codes. Then, we derive the sufficient and necessary conditions to determine the recoverability of erasure patterns for *ED-2* codes, EVENODD, and RDP. Practically, we propose an efficient decoding algorithm called *SCAN* algorithm to correct any erasure patterns for these codes.

The *SCAN* algorithm consists of two steps, *Check* operation and *Recover* operation. The *Check* operation efficiently determines the recoverability of an erasure pattern and builds recovery plan. Inheriting the result from the *Check* operation, the *Recover* operation recovers all erasures by performing only XOR operation. Hence, the *SCAN* algorithm does not perform any unnecessary operations for failure recovery.

The decoding performance of the *SCAN* algorithm is evaluated by comparisons with another generic decoding algorithm *Matrix Method*. The evaluation is performed on three erasure codes: X-code, EVENODD, and RDP, and it covers all possible disk failure types. Extensive experiments and analysis show that the *SCAN* algorithm outperforms the *Matrix Method* for all disk failure types and on all evaluated erasure codes. Therefore, the *SCAN* algorithm has a great potential to be integrated into practical RAID-6 arrays/clusters.

A possible future research direction is to extend these results for RAID-6 to RAID-n ($n \geq 7$) systems.

CHAPTER 4 Efficient Error Decoding for the STAR Code

4.1 Introduction

As storage systems grow in size and complexity, various hardware and software component failures inevitably occur [86, 98, 12, 67], which often result in various user data losses and errors [14, 13, 12]. From an application's point of view, a data loss can be either *failure* or *silent error*. Here a failure refers to a data loss with explicit error report from a disk drive to the whole system or application. The simplest and most common one is an entire disk drive failure in a *fail-and-stop* fashion, resulting in whole data loss on the disk. Another failure case is the *latent sector failure* within a disk drive [13], which can also be detected and reported during the *scrubbing* process using a disk driver's internal *error correcting codes*. A silent error, however, refers to a data *corruption* without any error indication from the disk drive itself to the system. This type of error includes *corrupted data*, *torn writes*, *lost writes*, *misdirected writes* and *wrong reads*, which usually results from various bugs in the firmware on disk controllers and various other related software in the storage stacks along a data I/O path [95, 12, 67, 48, 34].

Failures have been known for a long time, and various techniques have been developed to cope with them, from data mirroring to simple replication to more advanced *error control code* based RAID-5 and RAID-6 type of data protection schemes [85, 25, 40, 20, 31]. In contrast, silent errors are less well-understood and not as sufficiently accounted for. By detecting and converting silent errors into failures, *checksum* at disk sector or block level is an effective and the most common technique to combat silent errors. Unfortunately, checksum is not sufficient by itself. For example, a silent error from a write to a wrong sector or block due to a firmware bug cannot be detected by the checksum on the very sector or block at all. As a result, various bandits using file system level information have been proposed as additional measures to deal with silent errors. These proposals include *write verification*, *physical and logical identity*, and *version mirroring* [12, 67]. Seemingly different individually, these techniques are common in their implicit premise – adding redundant information can help detect errors.

From a different perspective, however, redundancy is already in place introduced by error correcting

codes to cope with failures at system level. From coding theory point of view, a disk failure corresponds to an *erasure* in corresponding codeword, while a silent error corresponds to an *error*. A proper error correcting code can correct certain numbers of both erasures and errors at the same time. Thus we advocate using error correcting codes and overcome both failures and silent errors simultaneously as a more unified and systematic mechanism to simplify storage system design. Specifically, we recommend the *STAR* code [54] as a very suitable candidate for such purpose. The STAR code was recently introduced to tolerate *three* simultaneous disk failures in a storage system. While initially designed to tolerate only disk failures, as an MDS code with minimum distance of 4, it can also protect *simultaneous* failures on one disk and silent errors in another disk [79]. While any MDS code with minimum distance of 4 can achieve this goal, the STAR code is unique in its geometric structure, which allows us to design a special decoding algorithm that can recover failures and errors very efficiently. Therefore, the very *focus* of this chapter is to provide such a decoding algorithm so that the STAR code can be used to effectively and efficiently deal with failures and silent errors at the same time, and thus significantly improve the data reliability of storage systems.

In a storage system, the decoding algorithm can be used in two situations. One situation is during an *inter-disk scrubbing* process. Different from *intra-disk* scrubbing process which uses checksums at sector or block level to detect disk errors, an *inter-disk* scrubbing process collects data from multiple disks and check the data consistency according to certain constraints, such as the parity constraint imposed by the STAR code. In this situation, even if failures are present, the decoding algorithm can still detect silent errors and correct them. The second situation is in a data reconstruction process. When one disk completely fails, temporal correlations can result in the probability of silent errors in other similar disks used in the same system much higher than normal [12, 67]. Hence, when recovering the disk failure, using the decoding algorithm to perform error detection and correction at the same time can achieve much higher data reliability.

The main contributions of this chapter include: 1) the design of an efficient decoding algorithm named *EEL* (Efficient Error Locating) for the STAR code to tolerate one failure and one silent error at the same time; 2) a rigorous correctness proof of the decoding algorithm; and 3) performance evaluation of the decoding algorithm with thorough comparisons to a naive try-and-test decoding algorithm for the same purpose.

The rest of the chapter is organized as follows: Section 4.2 discusses related work; Section 4.3 lists related coding theory terms and results, and gives a brief description of the STAR code; Section 4.4 then discusses how to detect errors for the STAR code; a naive try-and-test approach for the STAR code is then described in Section 4.5; Section 4.6 presents this chapter's main result: a new efficient algorithm for the STAR code to decode simultaneous failures and errors, with its computation performance evaluated in Section 4.7;

Some broader applications of the STAR code together with the new decoding algorithm are discussed in Section 4.8; and Section 4.9 concludes the chapter.

4.2 Related Work

Directly *related* to this work is certainly the STAR code design [54]. The STAR code is designed to tolerate three disk failures, and it has shown to have much better encoding/decoding performance than other similar codes [54]. The decoding algorithm presented in [54], however, can only correct up to three disk *failures* (erasures). Similarly, only erasure decoding algorithm has been presented for the recently introduced generalized RDP code, another MDS code with minimum distance of 4 [19]. In general coding theory, though, various decoding algorithms have been designed to correct erasures and errors at the same time for certain codes, such as the BCH code and the Reed-Solomon code [79, 111]. However, there is *no* general erasure-and-error decoding algorithm for any code, except the naive one which will be described and compared later in Section 4.5.

From the storage system point of view, various techniques and systems have been developed to combat disk failures [85, 25, 40, 20, 31]. Also as already described and discussed in [101, 95, 12, 67], various techniques and schemes have been developed to cope with silent errors. The results presented in this chapter complement all these techniques and schemes at the disk and system levels, including intra-disk scrubbing. In fact, use of the STAR code with our 1-erasure-and-1-error decoding algorithm will address the *parity pollution* issue as raised in [67], where a silent error in a data block spreads to other data blocks through various parity (checksum) calculations. With the STAR code, silent errors can be corrected within their data blocks without further spreading to other blocks.

4.3 Basics of the STAR Code

In this section, we give a brief description of the STAR code. It is an array code, which is easier to be described in a two-dimensional array rather than a one-dimensional vector. First we define some notations for an array code.

4.3.1 Notations

Table 4.1 lists all the notations to be used in the rest of the chapter. A letter in lower case denotes a symbol or a value, such as $a_{i,j}$, and a letter in upper case denotes a column, such as C_j. All the symbols are

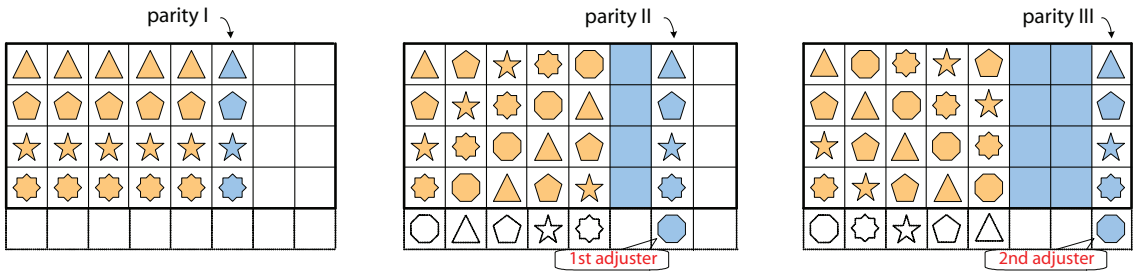


Figure 4.1: Construction of the STAR code

within one codeword.

Notation	Definition
$\langle x \rangle_p$	$x \ mod \ p$
$a_{i,j}$	original symbol at row i and column j
$c_{i,j}$	symbol read at row i and column j
C_j	column read (of symbols) at index j
$e_{i,j}$	error symbol at row i and column j
E_j	column (of error symbols) at index j
S_j	column (of syndromes) at index j
$\bigoplus_i a_i$	XOR all symbol a_i's (return one symbol)
$\bigoplus_j C_j$	XOR all column C_j's (return one column)
$C \bigoplus a$	XOR each symbol in column C with symbol a (return one column)
$C_i \bigoplus C_j$	XOR column C_i and C_j (return one column)
$f_{\downarrow}(C, i)$	cyclic shifting column C downward by i positions

Table 4.1: Notations Defined

Note that $a_{i,j}$ denotes an original symbol. It is the correct value in a codeword. $c_{i,j}$ denotes the symbol read from a disk. It may be unknown due to a disk failure or incorrect due to a silent error. $e_{i,j}$ denotes the error symbol causing $a_{i,j}$ flip to $c_{i,j}$. If it is 0, then there is no error; otherwise, the read symbol is corrupted with $c_{i,j} = a_{i,j} \oplus e_{i,j}$.

4.3.2 STAR code: A Brief Description

The STAR code can be described by a $(k-1) \times (k+3)$ 2-dimensional array. For best storage and computation performance, $k = p$, where p is a prime number, though a general STAR code for arbitrary k can be easily derived from its closest p [54, 92]. For simplicity, we only limit our discussion for $k = p$ throughout the chapter.

A STAR codeword consists of p user data columns and 3 parity columns. The $1st$ parity column is a *horizontal* parity column, which is calculated by XORing all the data symbols in the same row. The $2nd$ parity column is a *diagonal* parity column. Its computation is as follows. First, an *adjuster* is computed from the data symbols along the main diagonal of slope 1. Second, the data symbols along other slope 1 diagonals are computed as diagonal parity symbols. Third, the adjuster is complemented to all the diagonal parity symbols. The $3rd$ parity column follows a similar construction as the $2nd$ parity column, except that it is computed along diagonals of slope -1.

Figure 4.1 shows the construction of the STAR code for $p = 5$. Together with the $(5-1) \times (5+3)$ two dimensional array, the figure also contains an *imaginary* 5^{th} row, where all the data symbols are set to 0. It is shown only to help understand the adjuster and parity calculation. Without the $3rd$ parity column, the STAR code reduces to the $(p+2, p)$ EVENODD code [20]. The algebraic construction of the three parity columns is defined as follows ($0 \leq i < p-1$):

$$a_{i,p} = \bigoplus_{j=0}^{p-1} a_{i,j};$$

$$a_{i,p+1} = t_1 \oplus \left(\bigoplus_{j=0}^{p-1} a_{<i-j>_p,j} \right), \text{where } t_1 = \bigoplus_{j=0}^{p-1} a_{<-1-j>_p,j};$$

$$a_{i,p+2} = t_2 \oplus \left(\bigoplus_{j=0}^{p-1} a_{<i+j>_p,j} \right), \text{where } t_2 = \bigoplus_{j=0}^{p-1} a_{<-1+j>_p,j}.$$

Here, t_1 and t_2 are the *adjusters* for the $1st$ and $2nd$ diagonal parity columns, respectively.

When a column is treated as a *super* symbol, the STAR code is a $(p+3, p)$ code, and its minimum (column) distance is four [54]. (When used in storage systems, a column usually is mapped to a disk drive.) An efficient decoding algorithm for recovering three erasures for the STAR code was presented in [54]. In this chapter, however, our focus is on how to simultaneously correct one erasure and one error for the STAR code.

4.4 Error Detection for the STAR code

In general, correcting a codeword with errors involves two steps: *error detection* and *error correction*. The error detection step determines whether there is any error in a codeword, and, if so, the error correction step is invoked to correct the error. Depending on the positions of erasure and error, both the detection and

correction algorithms vary slightly. Nevertheless, the essence stays the same. Hence, for illustration purpose, we first focus on one single error type here, namely, *both the erasure and error columns are among the data columns*. A complete decoding algorithm will be provided later in Sec. 4.6 to deal with all possible erasure and error locations.

Assume the erasure column is C_u. Let R_u^0 denote the column recovered from the horizontal parity column, R_u^1 from the $1st$ diagonal parity column and R_u^2 from the $2nd$ diagonal parity column. R_u^0, R_u^1 and R_u^2 can be calculated as follows:

$$R_u^0 = \bigoplus_{j=0, j\neq u}^{p} C_j;$$

$$R_u^1 = f_{\downarrow}(C_{p+1} \bigoplus \left(\bigoplus_{j=0, j\neq u}^{p-1} f_{\downarrow}(C_j, j) \right) \bigoplus r_u^1, -u),$$

$$\text{where } r_u^1 = \bigoplus_{j=0}^{p-1} c_{<-1-j+u>_p, j} \bigoplus c_{<-1+u>_p, p+1};$$

$$R_u^2 = f_{\downarrow}(C_{p+2} \bigoplus \left(\bigoplus_{j=0, j\neq u}^{p-1} f_{\downarrow}(C_j, -j) \right) \bigoplus r_u^2, u),$$

$$\text{where } r_u^2 = \bigoplus_{j=0}^{p-1} c_{<-1+j-u>_p, j} \bigoplus c_{<-1-u>_p, p+2}.$$

Although R_u^0, R_u^1 and R_u^2 are computed from different parity columns, they represent the same data column. Hence, we can simply compare them to detect whether there is an error in the codeword. If they are all equal, then there is no error. The erasure column can be simply recovered by setting it to R_u^0, and the decoding process completes. Otherwise, there must exist at least one error column.

	no error	$v = p$	$v = p+1$	$v = p+2$	$0 \leq v \leq p-1$
$S_0(\alpha)$	$u(\alpha)$	$u(\alpha) + v(\alpha)$	$u(\alpha)$	$u(\alpha)$	$u(\alpha) + v(\alpha)$
$S_1(\alpha)$	$\alpha^u u(\alpha)$	$\alpha^u u(\alpha)$	$\alpha^u u(\alpha) + v(\alpha)$	$\alpha^u u(\alpha)$	$\alpha^u u(\alpha) + \alpha^v v(\alpha)$
$S_2(\alpha)$	$\alpha^{-u} u(\alpha)$	$\alpha^{-u} u(\alpha)$	$\alpha^{-u} u(\alpha)$	$\alpha^{-u} u(\alpha) + v(\alpha)$	$\alpha^{-u} u(\alpha) + \alpha^{-v} v(\alpha)$

Table 4.2: Error Column Location v and the Syndromes

	no error	$v = p$	$v = p + 1$	$v = p + 2$	$0 \leq v \leq p - 1$
$SS_1(\alpha)$	0	$v(\alpha)$	$\alpha^{-u}v(\alpha)$	0	$v(\alpha) + \alpha^{v-u}v(\alpha)$
$SS_2(\alpha)$	0	$v(\alpha)$	0	$\alpha^{u}v(\alpha)$	$v(\alpha) + \alpha^{u-v}v(\alpha)$

Table 4.3: Error Column Location v and Simplified Syndromes

4.5 A Naive Decoding Algorithm: Try-and-Test

As there is no published erasure-and-error decoding algorithm to compare with, we first describe a simple decoding algorithm. The idea is straightforward: simply use a *try-and-test* approach – whenever an error is detected, the algorithm tests each of the survival columns sequentially until the error column is found. For each column being tested, the algorithm first treats it as another erasure column and then checks *parity consistency* of the remaining columns.

The parity consistency is checked as follows. Assume both u and v are data columns. u denotes the original erasure column and v denotes the tested error column. Then, all the three parity columns are available. The Try-and-Test approach uses the first two parity columns to decode column u and v, following the erasure decoding algorithm of the STAR code [54]. It then re-encodes the $3rd$ parity column from all the data columns, and compares it with the original one. If they are equal, then column v is indeed the error column, and as a byproduct, both columns u and v are recovered during this process. Otherwise, test the next column similarly until the error column is found or all the columns are tested. If all the columns are tested, but none of them can be deemed as an error, then there are more than one error column, and the decoding algorithm declares a decoding failure event.

The above parity consistency check can be readily generalized to other cases, where both column u and v are parity columns, or one is a data column while the other is a parity column. In addition, interested readers can easily prove that the Try-and-Test approach can indeed correct one erasure column and one error column for the STAR code. we simply skip these simple but tedious discussions here.

4.6 The *EEL* Algorithm

Now we propose a new decoding algorithm - the *EEL* (*Efficient Error Locating*) algorithm. The key is to locate the error column efficiently. The *EEL* algorithm leverages the unique intrinsic geometric structure of the STAR code and locates the error column without performing the Naive algorithm's *try-and-test* operation in locating the error column, which in turn greatly improves the decoding efficiency. Although the *EEL*

algorithm presented here is for *1-erasure-and-1-error* case, the algorithm can be simplified for *1-error* case. The details are left to interested readers again. First we show a basic error-locating algorithm and then we present a complete decoding algorithm with further improved computation performance.

4.6.1 A Basic Error-Locating Algorithm

Similar to the EVENODD code [21], let $A = [a_{i,j}]$ be a $(p-1) \times n$ binary array, with $a_{i,j}$ as defined in Section 4.3.1. Each column a_j $(0 \leq j \leq n-1)$ then can be viewed as a polynomial $a_j(\alpha)$ modulo $M_p(\alpha)$ [21, 54], where

$$a_j(\alpha) = a_{p-2,j}\alpha^{p-2} + \cdots + a_{1,j}\alpha + a_{0,j}$$

and

$$M_p(\alpha) = \frac{\alpha^p - 1}{\alpha - 1} = \alpha^{p-1} + \alpha^{p-2} + \cdots + \alpha + 1$$

Adopting the notation in [21], let $B = [a_{i,j}]$ be the error-corrupted array of A. Assume B has one erasure at column u, and at most one error at column v with the error value being $v(\alpha)$. (Note here u is known, but v is unknown.) Write

$$B = (b_0(\alpha), b_1(\alpha), \cdots, b_{n-1}(\alpha))$$

then for the STAR code, its three syndromes can be computed as follows:

$$\begin{aligned} S_0(\alpha) &= \bigoplus_{i=0}^{p} b_i(\alpha) \\ S_1(\alpha) &= b_{p+1}(\alpha) \oplus \left(\bigoplus_{i=0}^{p-1} \alpha^i b_i(\alpha) \right) \\ S_2(\alpha) &= b_{p+2}(\alpha) \oplus \left(\bigoplus_{i=0}^{p-1} \alpha^{-i} b_i(\alpha) \right) \end{aligned}$$

It is not hard to derive the relation between the error column location v and the above three syndromes, as shown in Table 4.2.

We further define two *simplified* syndromes as

$$\begin{aligned} SS_1(\alpha) &= S_0(\alpha) + \alpha^{-u} S_1(\alpha) \\ SS_2(\alpha) &= S_0(\alpha) + \alpha^{u} S_2(\alpha) \end{aligned}$$

and Table 4.2 can then be simplified to Table 4.3, from which the error column location can be easily

deduced by comparing the two simplified syndromes. Just note, when $0 \leq v \leq p-1$, i.e., the error is in a data column, the error location v is the first v satisfying the equation $\alpha^{u-v} SS_1(\alpha) = SS_2(\alpha)$.

From the above observations, an efficient erasure-and-error decoding algorithm can be readily derived for the STAR code. It is worthy noting, however, that all the above polynomial operations are performed using modulo $M_p(\alpha)$, which forces a multiplication by α take the *complement* of the shifted column of size $p-1$ [21, 54], just as the adjusters t_1 and t_2 computation in Sec. 4.3.2.

A natural question to ask is then: can this complement operation be avoided to further improve the decoding performance? Fortunately, the answer is positive. Through the rest of this section, we present a more efficient decoding algorithm, in which all the polynomial operations are performed over modulo $\alpha^p - 1$ instead, thus avoiding all the complement operations. To make the implementation of this algorithm more straightforward, vectors will be used hereafter, following the notations in Table 4.1.

	0	1	2	3	4	S_0	S_1	S_2
0		u_0	v_0			u_0+v_0	$v_3+u_3+v_2$	$u_1+v_2+u_0+v_1$
1		u_1	v_1			u_1+v_1	$u_0+u_3+v_2$	$u_2+v_3+u_0+v_1$
2		u_2	v_2			u_2+v_2	$u_1+v_0+u_3+v_2$	$u_3+u_0+v_1$
3		u_3	v_3			u_3+v_3	$u_2+v_1+u_3+v_2$	$v_0+u_0+v_1$
4								

(a) Step 1: compute syndrome

	0	1	2	3	4	S_0	S_1	S_2
0		u_0	v_0			u_0+v_0	$v_3+u_3+v_2$	$u_1+v_2+u_0+v_1$
1		u_1	v_1			u_1+v_1	$u_0+u_3+v_2$	$u_2+v_3+u_0+v_1$
2		u_2	v_2			u_2+v_2	$u_1+v_0+u_3+v_2$	$u_3+u_0+v_1$
3		u_3	v_3			u_3+v_3	$u_2+v_1+u_3+v_2$	$v_0+u_0+v_1$
4							u_3+v_2	u_0+v_1

(b) Step 2: compute adjuster syndrome

	0	1	2	3	4	S_0	S_1	S_2
0		u_0	v_0			u_0+v_0	v_3	u_1+v_2
1		u_1	v_1			u_1+v_1	u_0	u_2+v_3
2		u_2	v_2			u_2+v_2	u_1+v_0	u_3
3		u_3	v_3			u_3+v_3	u_2+v_1	v_0
4							u_3+v_2	u_0+v_1

(c) Step 3: cancel adjuster syndrome

	0	1	2	3	4	S_0	S_1	S_2
0		u_0	v_0			u_0+v_0	v_0	v_0+v_1
1		u_1	v_1			u_1+v_1	v_0+v_1	v_1+v_2
2		u_2	v_2			u_2+v_2	v_1+v_2	v_2+v_3
3		u_3	v_3			u_3+v_3	v_2+v_3	v_3
4							v_3	v_0

(d) Step 4: cancel erasure symbols

Figure 4.2: Locating error column in the *EEL* algorithm.

4.6.2 Syndrome Computation

The *EEL* algorithm consists of three steps: computing syndrome, locating error column, and recovering erasure and error columns. Recall from Section 4.3.2, as it has three parity constraints, the STAR code has three syndromes, and they can be computed as follows, where syndrome S_0 represents the horizontal parity constraint, S_1 the $1st$ diagonal parity constraint, and S_2 the $2nd$ parity constraint.

$$S_0 = \bigoplus_{j=0}^{p} C_j;$$

$$S_1 = C_{p+1} \bigoplus \left(\bigoplus_{j=0}^{p-1} f_{\downarrow}(C_j, j) \right) \bigoplus t_1,$$

$$\text{where } t_1 = \bigoplus_{j=0}^{p-1} c_{<-1-j>_p, j};$$

$$S_2 = C_{p+2} \bigoplus \left(\bigoplus_{j=0}^{p-1} f_{\downarrow}(C_j, -j) \right) \bigoplus t_2,$$

$$\text{where } t_2 = \bigoplus_{j=0}^{p-1} c_{<-1+j>_p, j}.$$

We note that the syndrome computation is time consuming, as its complexity is in the order of p^2. Fortunately, it is possible to leverage the results produced in the error detection step and thus greatly speedup this computation. In the rest of this section, we focus on the most common and hardest erasure and error pattern, where both the erasure and error are data columns. Again, u denotes the index of the erasure column, and no data is available from column u. Hence, we set $C_u = 0$.

We obtain the following relationship between the syndromes S_j's and R_j's, the results from the error detection step:

$$S_0 = R_u^0;$$

$$S_1 = f_{\downarrow}(R_u^1 \bigoplus r_u^1 \bigoplus t_1, u);$$

$$S_2 = f_{\downarrow}(R_u^2 \bigoplus r_u^2 \bigoplus t_2, -u).$$

Here, t_1, t_2, r_u^1, and r_u^2 are calculated similarly as in the syndrome computation.

4.6.3 Locating the Error Column

An Example

Locating the error column is the key step in our decoding algorithm. It is instructive to use an example to demonstrate how this step works.

Again the erasure u and error v are both data columns. For the error column v, we know that each symbol equals the XOR sum of the corresponding original data symbol and error symbol; that is, $c_{i,v} = a_{i,v} \oplus e_{i,v}$. For simplicity, denote $e_{i,v}$ by v_i. For the erasure column u, we set $C_u = 0$, so $a_{i,u} = e_{i,u}$ since $a_{i,u} = e_{i,u} \oplus c_{i,u}$ and $c_{i,u} = 0$. Similarly, denote $a_{i,u}$ by u_i. In the example shown in Figure 4.2, $u = 1$ and $v = 2$; $+$ denotes the XOR operation. Note here u_i's, v_i's (the values of the erasure and error columns) and v (the

error column location) are unknown, but $u = 1$ (the erasure column location) is known.

The error column can be located in the following five steps with explanations:

Step I – compute syndrome. We first show the relationship between symbols in the syndromes and the erasure/error columns. Using the first symbol of S_0 as an example (denoted as $S_{0,0}$). From Section 4.6.2, we know that $S_{0,0} = \oplus_{j=0}^{5} c_{0,j}$. For columns $j = 0, 3, 4, 5$ (neither erasure nor error), $c_{0,j} = a_{0,j}$. On the other hand, from the STAR code's encoding rule, $a_{0,5} = \oplus_{j=0}^{4} a_{0,j}$. Thus, by simple substitution, we get $S_{0,0} = u_0 \oplus v_0$. The rest of the syndrome symbols can be derived similarly, as shown in Figure 4.2(a).

Step II – compute adjuster. Now we simplify the syndromes for further calculation. In particular, the *adjuster* for the $1st$ diagonal parity column can be computed by XORing all the symbols in syndrome column S_0 and S_1, as shown in [20, 54]. We place the *adjuster* for S_1 at the last row in S_1. The adjuster for the $2nd$ diagonal parity column can be computed similarly. The results are shown in Figure 4.2(b).

Step III – complement adjuster. Then, we XOR the *adjuster* in S_1 with the rest of the symbols in S_1. This cancels the *adjuster* from the rest of the syndrome column S_1. The same operation is applied on S_2. Figure 4.2(c) shows the results.

Step IV – cancel erasure symbols. Now we cancel the symbols of the erasure column from S_1 and S_2, respectively. For S_1, this is achieved by shifting S_1 downwards by $-u$ (or 4) positions and XORed with S_0. For S_2, it is shifted downwards by $u = 1$ positions before XORed with S_0. All shifts are cyclic modular 5. The results are shown in Figure 4.2(d).

Step V – locate error column. The last step is to locate the error column. We observe that, if S_1 is cyclically shifted downwards by 4 positions, it matches S_2 exactly. In fact, the number of shifts (denoted as shift down position or h) is completely determined by the positions of the erasure and error column. It satisfies the following equation:

$$h = -v \text{ (error position)} + u \text{ (erasure position) (mod p)}$$

In this example, $u = 1$, $h = 4$ and $p = 5$, so $v = (u - h) = (1 - 4)$ (mod 5) $= 2$ and the error column is finally located!

The Error Locating Algorithm

The above example illustrates how to locate the error column. The error locating algorithm follows the same steps as those given in the example. Thus, the algorithm is quite straightforward. Below provides a

formal description of the algorithm together with its correctness proof.

Algorithm 6 Locating Error Column

/*Step 1: Compute syndromes S_0, S_1, S_2*/

$S_0 = R_u^0$;
$S_1 = f_\downarrow(R_u^1 \bigoplus r_u^1 \bigoplus t_1, u)$;
$S_2 = f_\downarrow(R_u^2 \bigoplus r_u^2 \bigoplus t_2, -u)$;

/*Step 2: Compute adjusters t_1, t_2*/

$t_1 = \bigoplus(S_0 \bigoplus S_1)$;
$t_2 = \bigoplus(S_0 \bigoplus S_2)$;

/*Step 3: Complement adjusters to the syndromes*/

$S_1 = S_1 \bigoplus t_1$;
$S_2 = S_2 \bigoplus t_2$;

/*Step 4. Cancel erasure symbols in the syndromes*/

$S_1 = S_1 \bigoplus t_1$;
$S_2 = S_2 \bigoplus t_2$;

/*Step 5. Compute the error column location*/

$f_\downarrow(S_1, -v + u) = S_2$.

In Algorithm 6, Step 1 uses some notations defined in Sec. 4.3.1 and Sec. 4.6.2. Note R_u^1, r_u^1, R_u^2 and r_u^2 are from the error detection process in Sec. 4.6.2.

To compute the error column location, Step 5 shifts S_1 down until S_1 = S_2. Again, shifts are cyclic modular p.

Let the number of shifts be h, then $h = -v + u \pmod p$. Solve v and it is the error column location.

If after p shifts, no S_1 matches S_2, then declare a decoding failure, indicating there are more than one error columns.

Now we prove the correctness of this error locating algorithm.

Proof. In the above algorithm, after Step 1, from Sec. 4.6.2 and Sec. 4.6.2, the following equations hold:

$$S_0 = E_u \bigoplus E_v, \tag{4.1}$$

$$S_1 = f_\downarrow(E_u, u) \bigoplus f_\downarrow(E_v, v) \bigoplus e_{p-1-u,u} \bigoplus e_{p-1-v,v},$$

$$S_2 = f_\downarrow(E_u, -u) \bigoplus f_\downarrow(E_v, -v) \bigoplus e_{p-1+u,u} \bigoplus e_{p-1+v,v}.$$

After Step 2, the following then becomes true by the definition of the adjusters:

$$t_1 = \bigoplus(S_0 \bigoplus S_1) = e_{p-1-u,u} \bigoplus e_{p-1-v,v},$$

$$t_2 = \bigoplus(S_0 \bigoplus S_2) = e_{p-1+u,u} \bigoplus e_{p-1+v,v}.$$

Hence after Step 3, combining the results from Step 1 and Step 2, we get

$$S_1 = f_{\downarrow}(E_u, u) \bigoplus f_{\downarrow}(E_v, v),$$

$$S_2 = f_{\downarrow}(E_u, -u) \bigoplus f_{\downarrow}(E_v, -v).$$

Then with Step 4, we have

$$\begin{aligned} S_1 &= S_0 \bigoplus f_{\downarrow}(S_1, -u) \\ &= S_0 \bigoplus f_{\downarrow}(E_u, 0) \bigoplus f_{\downarrow}(E_v, v-u) \\ &= S_0 \bigoplus E_u \bigoplus f_{\downarrow}(E_v, v-u) \\ &= E_u \bigoplus E_v \bigoplus E_u \bigoplus f_{\downarrow}(E_v, v-u) \\ &= E_v \bigoplus f_{\downarrow}(E_v, v-u). \end{aligned} \tag{4.2}$$

$$\begin{aligned} S_2 &= S_0 \bigoplus f_{\downarrow}(S_2, u) \\ &= S_0 \bigoplus f_{\downarrow}(E_u, 0) \bigoplus f_{\downarrow}(E_v, -v+u) \\ &= S_0 \bigoplus E_u \bigoplus f_{\downarrow}(E_v, -v+u) \\ &= E_u \bigoplus E_v \bigoplus E_u \bigoplus f_{\downarrow}(E_v, -v+u) \\ &= E_v \bigoplus f_{\downarrow}(E_v, -v+u). \end{aligned} \tag{4.3}$$

For the right part of Eq (4.2) and (4.3), we have:

$$\begin{aligned} &f_{\downarrow}(E_v \bigoplus f_{\downarrow}(E_v, v-u), -v+u) \\ &\quad = E_v \bigoplus f_{\downarrow}(E_v, -v+u) \end{aligned}$$

Hence $f_{\downarrow}(S_1, -v+u) = S_2$.

4.6.4 Recovering Erasure and Error Columns

After the error column is located, the next step is to correct both erasure and error columns. An intuitive approach is to treat the error column as another erasure column, then recover them using the STAR erasure decoding algorithm [54]. This naive approach, however, is not efficient as it does not utilize the intermediate results produced during the above error locating process. Now we present a much more efficient algorithm which can correct the erasure and error columns directly by fully utilizing those intermediate results. We first illustrate the algorithm by completing the previous example and then give a formal description of the algorithm itself.

An Example

We continue the example in Section 4.6.3 to demonstrate how the correcting algorithm works. In Figure 4.2(d), there are 5 rows in syndrome S_1. We can treat each row as an equation and each error symbol as a variable. Thus, there are totally 5 equations and 4 variables. In row 4, there is only one variable v_3, so we can compute v_3 from the equation represented by row 4. In row 3, there are two variables v_3 and v_2. After v_3 is solved, we can calculate v_2. Following the similar steps, we can solve v_1 from row 2 and v_0 from row 1. Now, all the error symbols in error column 2 are corrected.

The next step is to recover the erasure column 1. In syndrome S_0, there are four rows, and each row is the XOR sum of the symbols from erasure column 1 and error column 2. Each row is again treated as an equation. Since all the error symbols of column 2 are now known, we can compute u_0 from row 0 of S_0, u_1 from row 1, u_2 from row 2, and u_3 from row 4. All the erasure symbols of column 1 are thus recovered.

Finally, the error correction process completes by XORing the error symbols of column 2 with C_2 to recover the original data column 2.

The Erasure and Error Recovery Algorithm

Now we present a formal description of the erasure and error recovery algorithm. The pseudocode of the algorithm is described in Algorithm 7.

In Step 1, we solve p variables v_i's ($0 \leq i \leq p-1$) from a group of p linear equations. Observe that

1. each equation is the XOR sum of two variables, one from E_v and the other from $f_{\downarrow}(E_v, v-u)$;
2. each variable v_i appears exactly twice in the equations;
3. $v_{<-1>_p} = 0$, since it is in the imaginary row $p-1$.

Algorithm 7 Recovering erasure and error columns.

/*Step 1: Solve E_v, the error symbols*/
$E_v \bigoplus f_{\downarrow}(E_v, v-u) = S_0 \bigoplus f_{\downarrow}(S_1, -u)$;

/*Step 2: Recover E_u, the erasure column*/
$E_u = S_0 \bigoplus E_v$;

/*Step 3: Recover the original data column C_v*/
$C_v = C_v \bigoplus E_v$.

Hence the above equations in turn can be efficiently solved in a zig-zag fashion just as in the erasure decoding for the EVENODD code [20] by the following 3 steps:

1. Step 1a: start from the last row of Eq (4.2), which contains two variables $v_{<-1>_p}$ from E_v and $v_{<-1-1*(v-u)>_p}$ from $f_{\downarrow}(E_v, v-u)$. Since $v_{<-1>_p} = 0$, then the only one unknown variable $v_{<-1-1*(v-u)>_p}$ is solved. And now we are at row $\langle -1 - l*(v-u)\rangle_p$ (l=0). Go to Step 1b.

2. Step 1b: row $\langle -1 - l*(v-u)\rangle_p$ of Eq (4.2) consists of two variables $v_{<-1-l*(v-u)>_p}$ from E_v and $v_{<-1-(l+1)*(v-u)>_p}$ from $f_{\downarrow}(E_v, v-u)$. Since variable $v_{<-1-l*(v-u)>_p}$ is known, the only one unknown $v_{<-1-(1+1)*(v-u)>_p}$ can be recovered.

3. Step 1c: in step 1b we are at row $v_{<-1-l*(v-u)>_p}$. If $l = p-2$, this process stops; otherwise, go to the next row $\langle -1 - (l+1)*(v-u)\rangle_p$ and repeat step 1b.

□

4.7 Performance Evaluation

Now we evaluate the computation performance of our decoding algorithm by comparing it with the naive decoding algorithm described in Section 4.5. We first count the number of XOR needed for both algorithms, since XOR is the most frequent operation in decoding and thus dominates decoding performance. Then we measure the wall time of the decoding algorithms through experiments.

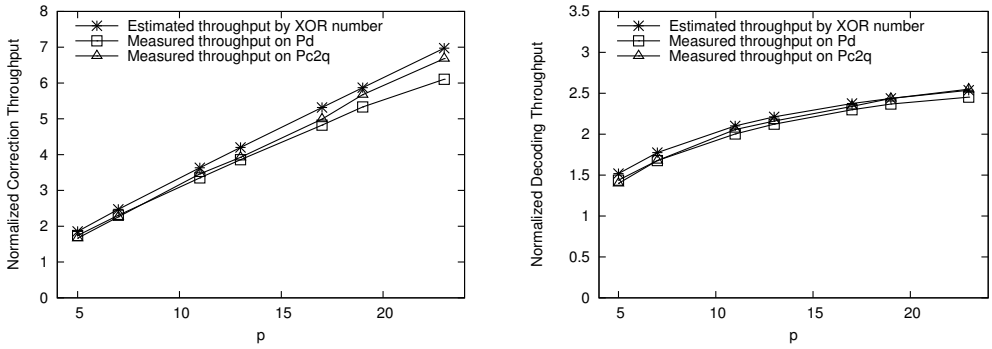
4.7.1 XOR Numbers

There are six possible erasure and error pattern combinations, as listed in the first two columns in Table 4.4. For both decoding algorithms, the XOR number needed to correct different pattern is different. Therefore, we first compute the number of XORs needed for each pattern. Then, by assuming each column has

Erasure	Error	Error Detection	Correction (*EEL*)	Total (*EEL*)	Correction (Naive)	Total (Naive)
Data	No	$3p^2 - 3p$	0	$3p^2 - 3p$	0	$3p^2 - 3p$
Parity	No	$2p^2 - 2p$	$p^2 - p$	$3p^2 - 3p$	$p^2 - p$	$3p^2 - 3p$
Data	Data	$3p^2 - 3p$	$21p - 16$	$3p^2 + 18p - 16$	$6.5p^2 - 6.5p$	$9.5p^2 - 9.5p$
Data	Parity	$3p^2 - 3p$	$20p - 15$	$3p^2 + 17p - 15$	$13p^2 - 22p$	$16p^2 - 25p$
Parity	Data	$2p^2 - 2p$	$p^2 + 14p - 13$	$3p^2 + 12p - 13$	$4.5p^2 - p$	$6.5p^2 - 3p$
Parity	Parity	$2p^2 - 2p$	$p^2 + 4p - 5$	$3p^2 + 2p - 5$	$8p^2 - 8p$	$10p^2 - 10p$

Table 4.4: Decoding cost comparison (in XORs).

the same probability to be an erasure or an error column, we calculate the *average* number of XORs as the cost for a decoding algorithm.



(a) Correction throughput of the *EEL* algorithm (b) Overall decoding throughput of the *EEL* algorithm

Figure 4.3: Comparison of throughput.

XOR number for the *EEL* algorithm

Table 4.4 shows that there are four possible erasure and error patterns when there is an error. We stick to the most common one - an erasure and an error in two data columns - to demonstrate how to count the number of XORs needed for decoding. The XOR numbers for the other patterns can be counted similarly.

Recall that there are in total four steps in our EEL decoding algorithm: 1) error detection, 2) syndrome computation, 3) error locating and 4) erasure and error recovery.

1. As shown in Section 4.4, the *error detection* step computes R_u^0, R_u^1 and R_u^2, each costing $p * (p - 1)$ XORs. Hence the total cost for this step is $3p^2 - 3p$. Note the symbols in the last imaginary row are not involved in computation.

2. As discussed in Section 4.6.2, the *syndrome computation* step computes syndromes S_0, S_1 and S_2 from R_u^0, R_u^1 and R_u^2 obtained in the error detection step. S_0 needs $p - 1$ XORs, and S_1 and S_2 each need $4p - 2$ XORs. Hence the total cost of this step is $9p - 5$.

3. The *error locating* step first simplifies syndromes S_1 and S_2; each computation needs $4p - 3$ XORs. Then vector *equivalence test* is conducted using *shift* and *compare*. In the test, however, no XOR operation is needed. As a result, the total cost for this step is only $8p - 6$ XORs.

4. The *erasure and error recovery* step needs $2p - 3$ XORs in solving the erasure symbol E_u and the error symbol E_v, and then $2p - 2$ XORs for recovering the erasure column E_u and error column E_v; hence, $4p - 5$ XORs in total.

Adding all the XOR numbers in the above steps, we get the decoding cost, which is $3p^2 + 18p - 16$ XORs. Similar analysis can be conducted for the other erasure and error patterns, as listed in details in Table 4.4. If a number contains a constant value less than 5, the constant value is ignored. In Table 4.4, The first two columns specify an erasure and error pattern. In the second column, a no means there is no error. The 3^{rd} column is the XOR numbers needed in the error detection step; the 4^{th} column is the number of XORs for the erasure and error correction step; the 5^{th} column is the total number of XORs performed in decoding process.

XORs Needed for the Naive algorithm

The Naive algorithm also consists of two steps, the *error detection* and the *error correction*. The error detection step is exactly the same as in our decoding algorithm, so we just focus on the error correction step.

We observe that as in the *EEL* algorithm, the Naive algorithm can greatly reduce XORs needed in the error correction step by utilizing the results from the error detection step. When the erasure column is a data column, the cost to test whether a data column is an error or not is $13p - 13$ when the results from the error detection step are used. Otherwise that cost would be $3p^2 - 3p$. So the cost reduction is significant. Assume that the average number of *try and test* is $p/2$ since there are in total p data columns, then the total average cost is $(13p - 13) * p/2 = 6.5p^2 - 6.5p$ XORs for correcting when both the erasure and error are data columns. When the error is a parity column, $(13p - 13) * (p - 1)$ XORs are needed on the $p - 1$ data columns, and $4p$ tests on the parity columns, hence the total cost is then $13p^2 - 22p$ XORs. Costs for correcting other erasure and error patterns can be counted similarly, as again listed in details in Table 4.4.

Comparison

When there is one erasure but no error, Table 4.4 shows that the *EEL* algorithm performs the same as the Naive algorithm, since both only conduct the same error detection step without invoking error correction.

The real comparison is when there is one error. The table shows that the *EEL* algorithm greatly outperforms the Naive algorithm.

4.7.2 Measured Decoding Performance

Experiment Setup

We have implemented both decoding algorithms in C language. We then measure their decoding times with random codewords. Codewords are kept in main memory, so there is no disk I/O involved. For a ($p+3$, p) STAR code, there are in total $p+3$ columns in a codeword. There are thus $p+3$ possible locations for an erasure or an error, and $(p+3)^2$ combinations for one erasure and one error to occur in a codeword. Note here if the error and the erasure occur on the same column, then that column is treated as an erasure column. In each test, all the $(p+3)^2$ possible erasure-error patterns are decoded to measure the average decoding time, where the value of an error is randomly generated. Such tests are repeated 3,000 times to get stable decoding times to experimentally compare the performance of the two algorithms.

The experiments are conducted on two platforms, one with Intel Pentium Dual Core CPU (named Pd) and the other with Intel Pentium Core 2 Quad CPU (named Pc2d). Both platforms run 64-bit Linux. On both platforms, the decoding algorithms are compiled by gcc with $-O2$ flag, a common choice for the optimization flag [77]. We use $gettimeofday$ system call to capture the time consumed in decoding, and the time elapsed in the decoding process is used as the decoding time. The ratio of the standard deviation to the average decoding time is at most 5%.

We use an optimization technique employed in [92, 77], where a large packet size can greatly improve encoding/decoding performance. Packet size means how many bytes in one symbol, and large packet size provides good data locality when performing XOR operation. In our experiments, the packet size is set to be 512 bytes so that one symbol naturally maps to one sector in a hard disk.

Decoding Throughput Comparison

Instead of comparing absolute values of XOR numbers and decoding throughput, we normalize the performance of the *EEL* Algorithm by that of the Naive Algorithm. The throughput is defined as the *reciprocal* of the decoding time or the XOR number. Note that both algorithms employ the exact same *error detection* process, and both algorithms include two steps: *error detection* and *error correction*. The error correction step is invoked only when error is detected, which is the focus of this chapter.

The decoding throughput in the *error correction* step measured from experiments are plotted in Figure 4.3(a), together with their corresponding XOR numbers, where the X-axis is the parameter p of a $(p+3, p)$ STAR code, and the Y-axis is the normalized decoding throughput for the error correction step of the corresponding STAR code. On both test platforms, the throughput estimated by the XOR number nicely matches the measured ones in the experiments. As p increases, the normalized throughput of the *EEL* Algorithm also increases.

Finally the *overall* decoding throughput of the *EEL* Algorithm is shown in Figure 4.3(b), again normalized by that of the Naive Algorithm, together with the corresponding XOR number. The overall decoding operation includes both the error detection step and the error correction step. Again the normalized overall decoding throughput of the *EEL* Algorithm is always greater than one, and also increases as p increases.

4.8 Further Discussions

First of all, it is easy to see that a similar erasure-and-error decoding algorithm can be designed for the generalized RDP code with minimum distance of 4 [19], as they have very similar geometric structures. The details are left to interested readers.

It is then worth noting that even though our erasure-and-error decoding algorithm for the STAR code is presented in the context of hard disk drives, it can certainly be extended from disk drives to storage nodes in a cluster or cloud system. More importantly, the STAR code and our decoding algorithm can certainly be used in other storage media, such as the DRAMS (*dynamic random access memory*). A very recent study [99] shows memory error rates are much higher than previously reported, and errors include permanent and soft ones. A permanent error can be detected but cannot be recovered without replacing the memory itself, while soft errors are transient and hard to detect without using error correcting codes. It is easy to see that they are just like disk failures and silent disk errors. Error control codes are already existing in a single memory chip to cope with *bit* errors. If the STAR code together with our decoding algorithm is applied to a group of memory chips in a system, the *uncorrectable* error rate [99] in a memory system can be greatly reduced, and hence severe consequences on a system resulting from memory errors, such as system crashes and service disruptions, can be significantly mitigated. Looking forward, the same technique can also be applied to *solid state disks* (SSD) which consist of multiple flash memory chip packages [29].

Also as already mentioned, it is easy to see that our decoding algorithm can be used to correct only one error (without any erasure) for the STAR code. Details for this simpler case are left to the readers interested

in this topic.

4.9 Summary

This chapter presents an efficient decoding algorithm for the STAR code to *simultaneously* tolerate one whole disk failure and another silent disk error in a storage system. In addition to a correctness proof of the algorithm, both theoretical analysis and experimental measurement show our decoding algorithm can outperform the best naive decoding algorithm we can think of by large factors in overall decoding throughput, and more in the error correction process. With this decoding algorithm, the STAR code can be used to significantly improve a storage system's reliability by effectively and efficiently coping with whole disk failure and silent errors at the same time.

Our future work is to improve error detection performance. Although the *EEL* algorithm achieves much better error correction performance than the naive algorithm, the need of performing error correction is relatively rare given that the probability of silent disk errors is low. (Certainly, error correction performance is very important when errors are detected.) A more general operation in a storage system is error detection since it is on regular I/O path and performed more often. Therefore, improving error detection performance would have higher impact in storage system's performance.

CHAPTER 5 Efficient Implementations of Large Finite Fields $GF(2^n)$

5.1 Introduction

Finite fields are widely used in constructing error-correcting codes and cryptographic algorithms. For example, Reed-Solomon codes [96] are based on arithmetic operations in finite fields. Various cryptographic constructions, including the Diffie-Hellman key exchange protocol [32], discrete-log based cryptosystems (e.g., El-Gamal encryption [35], DSA signatures [66]), and schemes based on elliptic curves [82] are implemented in finite fields of large prime order. While practical implementations of error-correcting or erasure codes use small finite fields to achieve high-throughput encoding and decoding, cryptographic systems need considerably larger finite fields for high security guarantees.

In this chapter, we provide efficient implementations of arithmetic operations for finite fields of characteristic two, ranging from $GF(2^{32})$ to $GF(2^{128})$. The main reason is that finite fields within this range are very suitable for secure data storage applications and systems. Most storage systems today employ erasure coding based on small finite fields (e.g., $GF(2^8)$ or $GF(2^{16})$) to provide fault tolerance in case of benign failures (for instance, drive crashes). They achieve efficiency through the use of small finite fields, but they have not been designed to sustain adversarial failures. With the advent of cloud storage, offered by providers such as Amazon S3 and others, a whole host of new failure models need to be considered, e.g., mis-configuration, insider threats, software bugs, and even natural calamities. Accordingly, storage systems have to be redesigned with robustness against adversarial failures.

One direct consequence is that reasonably larger finite fields are needed to realize both fault tolerance and security of storage systems. In general, the larger a finite field is, more security it offers. Compared to general cryptographic operations, though, for data storage applications, a finite field of size $GF(2^{64})$ or $GF(2^{128})$ is considered to be large enough to achieve desired security degree, while not imposing too much computational cost for other operations, such as erasure coding for data reliability. Thus throughout this chapter, our focus will be on finite fields up to $GF(2^{128})$.

To efficiently implement operations over finite fields of the form we are interested in, we combine well

established techniques with novel optimizations. Currently, several methods are most commonly used for implementing finite field arithmetic. The *binary polynomial method* represents finite field elements as polynomials and translates field arithmetic operations into corresponding operations on polynomials. While addition and subtraction are extremely fast (as they can be implemented with exclusive-or operations), polynomial multiplication and division involve operations known to be inefficient in fields of large order (e.g., modular reduction modulo an irreducible polynomial or finding polynomial inverses). With additional optimizations (precomputation of lookup tables), the binary polynomial method is nevertheless very efficient in small fields. For operations on larger finite fields, the *extension field method* [113] uses precomputed tables in the base field and several techniques [56, 113] for extending small field arithmetic into larger field operations.

In our algorithms, we use the extension field arithmetic method together with novel optimizations for operations in the base field (of small size). We propose to use certain irreducible polynomial patterns that reduce the complexity of modular reduction, and then we specifically design an efficient division algorithm for those polynomial patterns in the base field. We also use precomputed log and antilog tables and a new method of caching table lookup results for optimizing the multiplication operation in the base field. Using these techniques, we provide several optimized implementations of multiplication and division operations in such fields, compare their performance on various platforms with that of best known arithmetic operations, and show the practical benefits of our optimizations.

To summarize, the contributions of this chapter include:

1. We survey the major efficient algorithms that could be used for implementing arithmetic operations over reasonably large finite fields, as needed by secure storage applications.

2. We provide several implementations of arithmetic operations in large finite fields of characteristic two by extending existing methods with newly proposed optimizations.

3. We extensively evaluate and compare different implementations on multiple platforms and show which ones perform best under specific conditions.

5.2 Related Work

A lot of cryptographic algorithms are based on finite field arithmetic [32, 35, 82, 66]. Finite fields used in the design of cryptographic primitives can be classified into three types: prime fields, binary fields, and

optimal extension fields. Prime fields can be represented by $\mathbb{F}_p$, where p is a prime number. Binary fields are fields of characteristic two $\mathbb{F}_{2^n}$ or $GF(2^n)$, where n is an integer number greater than 0. Optimal extension fields are fields of the form $\mathbb{F}_{p^n}$, where p is a prime number and n is an integer number that has to satisfy some restrictions with respect to p [11]. Due to differences in the algebraic structure of these finite fields, the arithmetic operations in different types of fields are also implemented differently. Guajardo et al. presented a survey of efficient software implementations for general field arithmetic [46]. In this chapter, we focus on binary fields.

There are efficient multiplication and division approaches for general binary fields. Lopez et al. [73] introduced several multiplication algorithms for $GF(2^n)$. The algorithms include the *right-to-left comb method*, the *left-to-right comb method*, and the *left-to-right comb method with windows of width* w. These algorithms have been shown to greatly outperform the traditional shift-and-add method [50], and they are among the fastest existing multiplication algorithms. Widely used efficient algorithms for division include the *Extended Euclidean Algorithm* and its two variants: the *Binary Extended Euclidean Algorithm* [81] and the *Almost Inverse Algorithm* [100]. These algorithms are adapted from the classical Euclidean algorithm. We will compare our newly proposed algorithms in this chapter with the above algorithms.

DeWin et al. [113] presented a fast software implementation of arithmetic operations in $GF(2^n)$. In their fast implementation, large finite fields are viewed as extensions of base field $GF(2^{16})$. In [51], similar algorithms with base field $GF(2^8)$ were developed. As the fast implementation was proposed before the algorithms of right-to-left comb method and its variants, DeWin et al. did not compare their performances. Additionally, the DeWin implementation was evaluated on a single finite field $GF(2^{176})$, and it is unknown how the presented performance results translate to other fields. This chapter tries to address these limitations, by evaluating the proposed algorithms in a more extensive way.

The previous work most relevant to this chapter is by Greenan et al. [44, 45]. Greenan et al. [44, 45] described a variety of table lookup algorithms for multiplication and division operations over $GF(2^n)$, and evaluated their performance on several platforms. They concluded that the performance of different implementations of finite field arithmetic highly depends on the underlying hardware and workload. Our work differs from theirs in two aspects. First, their table lookup algorithms were implemented in small finite fields, up to $GF(2^{32})$. Second, they did not perform a comparison with the right-to-left comb method or its variants, currently the fastest known algorithms for multiplication. In our work, we study finite fields from $GF(2^{32})$ to $GF(2^{128})$, and we compare the performance of our algorithms with the left-to-right comb method with windows of width w.

For small finite fields, the result of an arithmetic operation could be directly looked up from pre-computed tables [87]. The number of lookups depends on the table lookup algorithm. Huang et al. introduced a couple of efficient table lookup implementations [56]. Their implementations do not contain any conditional branches and modular operations. This way greatly improves the performance of table lookup. Although their algorithms are designed for finite fields up to $GF(2^{16})$, they are also useful for implementing large finite fields. We incorporate some of their techniques in our proposed algorithms introduced in Section 5.4.

Besides targeted for a general platform, an implementation can be developed for a particular platform. Then, the implementation can take advantage of the instructions available at that platform to achieve high performance [59, 62]. For example, Aranha et al. [9] introduced a new split form of finite field elements, and presented a constant-memory lookup-based multiplication strategy. Their approach made extensive use of parallel table lookup (PTLU) instructions to perform field operations in parallel. They have shown that their implementation is effective for finite fields from $GF(2^{113})$ to $GF(2^{1223})$ on the platforms supporting PTLU instructions. In this chapter, we have a different focus. We aim to optimize the performance of a general software implementation for finite fields.

There are several open source implementations for finite fields. One implementation is `relic-toolkit` provided by Aranha et al. [8]. `relic-toolkit` is a cryptographic toolkit that emphasizes efficiency and flexibility. It provides a rich set of functionalities, including prime and binary field arithmetic. Another library is `Jerasure`, implemented by Plank [93]. `Jerasure` supports erasure coding in storage applications. It implements finite fields from $GF(2^4)$ to $GF(2^{32})$, but it does not support larger ones. Nevertheless, `Jerasure` provides a good framework for finite field arithmetic, and we utilize it to develop our code for larger finite fields.

5.3 Arithmetic Operations in Finite Fields

In this chapter, we focus on arithmetic operations for large finite fields of characteristics two $GF(2^n)$, with $n = 16 * m$, such as $GF(2^{64})$ or $GF(2^{128})$, where field elements can be byte aligned. Most techniques presented in this chapter, however, can be readily applied to general finite fields $GF(2^n)$. In this section, we briefly introduce several well-known algorithms for arithmetic operations in finite fields, as well as their complexity analysis.

5.3.1 Binary Polynomial Method

According to finite field theory [46], elements of a finite field have multiple representations. In *standard basis* (or *polynomial basis*), an element of $GF(2^n)$ can be viewed as a polynomial $a(x) = a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \cdots + a_1x + a_0$ of degree $n-1$ with coefficients in $GF(2)$. The same element can be represented with a bit vector $(a_{n-1}, a_{n-2} \ldots, a_1, a_0)$ of length n. To generate efficient machine representations, bit vectors are grouped into multiple machine words. For instance, in a 64-bit machine, a single *long* value holds an element of finite field $GF(2^{64})$. Elements of larger fields are represented with multiple long values, e.g., two *long* values are used for one element in $GF(2^{128})$.

There are other field representations, e.g., using a *normal basis* [71]. A normal basis of $GF(2^n)$ is a basis of the form $(\beta, \beta^2, \ldots, \beta^{2^{n-1}})$, for some $\beta \in GF(2^n)$. An element in normal basis is represented as $b_{n-1}\beta^{2^{n-1}} + b_{n-2}\beta^{2^{n-2}} + \ldots b_1\beta^2 + b_0\beta$, where $b_i \in GF(2)$. The normal basis representation is efficient for speeding up exponentiations used in some cryptographic algorithms. In this chapter, however, we focus on the standard basis representation.

In the standard basis representation, addition and subtraction in $GF(2^n)$ can be simply implemented using bitwise XORs of bit strings of length n. To implement multiplication and division, we need to consider first an *irreducible polynomial* $f(x)$ of degree n over $GF(2)$ [71]. Then multiplication and division are defined as follows:

Multiplication of two polynomials $a(x)$ and $b(x)$: A simple multiplication algorithm is the classical shift-and-add method [100]. This method, however, is efficient in hardware, but not in software [46]. An efficient software implementation is the left-to-right comb method with windows of width w [73]. This algorithm first multiplies $a(x)$ and $b(x)$, resulting in a polynomial of degree at most $2n-2$. Then, the multiplication result is reduced modulo the irreducible polynomial $f(x)$ to obtain a polynomial in $GF(2^n)$. More details on this method are provided below. Other similar methods include the right-to-left comb method and the left-to-right comb method [73], but these methods have been shown to be slower than the left-to-right comb method with windows of width w [73].

We give now some details on the left-to-right comb method with windows of width w for multiplication. This method computes the multiplication of two polynomials $a(x)$ and $b(x)$ of degree at most $n-1$ over $GF(2)$. It is intuitively based on the observation that if $b(x) \cdot x^k$ is computed for a $k \in [0, W-1]$, where W is the machine word size, then $b(x) \cdot x^{Wj+k}$ can be computed by simply appending j zero words to the right of $b(x) \cdot x^k$ ([50, 73]). Furthermore, this method is accelerated significantly at the expense of a little

storage overhead. It first computes $b(x) \cdot h(x)$ for all polynomials $h(x)$ of degree at most $w-1$, and then it can process w bits of $a(x)$ at once rather than only one bit at a time. The pseudocode of this method is shown in Algorithm 8. In Algorithm 8, a, b, and c are coefficient vectors representing polynomials $a(x)$, $b(x)$ and $c(x)$. a is a vector of words of the form $(a[s-1], a[s-2], \cdots a[1], a[0])$, where $s = \lceil n/W \rceil$. Similar notations are used for b and c. One thing to note is that as Algorithm 8 runs, the length of c is $2s$, while the length of a and b is constant at s. More details of this method are available in [50, 73].

Algorithm 8 Left-to-right comb method with windows of width w

INPUT: Binary polynomials $a(x)$ and $b(x)$ of degree at most $n-1$ represented with vectors a and b, and $s = \lceil n/W \rceil$
OUTPUT: $c(x) = a(x) \cdot b(x)$ represented with vector c

```
1: Precompute b_h = b(x) · h(x) for all polynomials h(x) of degree at most w − 1
2: c ← 0;
3: for k from W/w − 1 to 0 do
4:    for j from 0 to s − 1 do
5:       Let h = (h_{w−1}, h_{w−2}, ..., h_1, h_0), where h_t is bit (wk + t) of a[j]
6:       for i from 0 to s − 1 do
7:          c[i + j] ← b_h + c[i + j]
8:       end for
9:    end for
10:   if k ≠ 0 then
11:      c ← c · x^w;
12:   end if
13: end for
```

In the multiplication operation, the left-to-right comb method with windows of width w is followed by a modular reduction step in which the degree of $c(x)$ is reduced from at most $2n-2$ to at most $n-1$. Generally, modular reduction for a random irreducible polynomial $f(x)$ is performed bit by bit, i.e., the degree of $c(x)$ is reduced by one in each step. However, if $f(x)$ is a trinomial or pentanomial (i.e., it has three or five non-zero coefficients, recommended by NIST in the standards for public key cryptography [36]), the reduction step can be efficiently performed word by word [46]. Then, the degree of $c(x)$ is reduced by W in one step, and the modular reduction of $c(x)$ is greatly sped up. In this chapter, we only use trinomial or pentanomial irreducible polynomials for finite fields ranging from $GF(2^{32})$ to $GF(2^{128})$, and therefore we perform the modular reduction of the multiplication result one word at a time.

Division of two polynomials $a(x)$ and $b(x)$: There are several different ways to implement the division operation. One method computes the inverse polynomial of $b(x)$ in $GF(2^n)$, denoted by $b^{-1}(x)$, and then multiplies $a(x)$ with $b^{-1}(x)$. Other methods directly compute the division result. Several of the popular division algorithms include the Extended Euclidean Algorithm, the Binary Extended Euclidean Algorithm

and the Almost Inverse Algorithm [50, 100]. These algorithms are adapted from the classical Euclidean algorithm [16].

Efficient division algorithms, including the Extended Euclidean Algorithm, the Binary Extended Euclidean Algorithm, and the Almost Inverse Algorithm, are all based on the Euclidean algorithm. Here, we briefly describe the idea behind the Extended Euclidean Algorithm. Assume $f(x)$ is an irreducible polynomial of degree n over $GF(2)$. For any $a(x)$ with coefficients in $GF(2)$, the Euclidean algorithm computes $\mathtt{gcd}(a(x), f(x)) = 1$ (since $f(x)$ is irreducible). Then, according to algebra theory [16],

$$\exists\, b(x),\ c(x) \text{ s.t. } a(x) \cdot b(x) + f(x) \cdot c(x) = 1 \bmod f(x) \tag{5.1}$$

The Extended Euclidean Algorithm computes both $b(x)$ and $c(x)$ in equation (5.1) when calculating $\mathtt{gcd}(a(x), f(x))$. It is easy to see that $a^{-1}(x) \bmod f(x) = b(x)$. Hence, the Extended Euclidean Algorithm computes $a^{-1}(x) \bmod f(x)$, the inverse of $a(x)$. Moreover, for fields of base 2, the Extended Euclidean Algorithm can be used to directly compute division without first obtaining $a^{-1}(x)$. The other two algorithms, the Binary Extended Euclidean Algorithm and the Almost Inverse Algorithm, are variants of the Extended Euclidean Algorithm optimized for $GF(2^n)$ [46].

As it is difficult to precisely analyze the time complexity of division, we instead provide the measured performance of division in Section 5.5. Interested readers can find the theoretical analysis of the Binary Extended Euclidean Algorithm in [107].

Using the standard basis representation, we implement the left-to-right comb method with windows of width w for multiplication and the Binary Extended Euclidean Algorithm for division. We refer to the use of these algorithms for implementing finite field arithmetic as the *binary polynomial method*.

5.3.2 Table Lookup Methods

There are various table lookup methods that precompute and store results of arithmetic operations in tables with the goal of speeding up evaluation of multiplication and division operations. These methods achieve tradeoffs between the amount of storage for precomputed tables and operation speed.

Full Multiplication and Division Tables

One simple table lookup method uses full multiplication and division tables. This algorithm precomputes the multiplication and division results for all element pairs in the field (by using, for instance, the binary

polynomial method described in Section 5.3.1) and stores the results in tables. The tables are kept in main memory for small fields. To perform a multiplication or division operation, this algorithm quickly looks up the result from the tables with no computation.

While this algorithms involves only one table lookup for both multiplication and division, its space complexity is quadratic in the size of the field. For $GF(2^n)$, its storage complexity is $(n/8) * 2^{2n+1}$ bytes. For most off-the-shelf machines, this memory requirement is acceptable for $GF(2^8)$, but not for larger finite fields. For example, full multiplication and division tables for $GF(2^{16})$ would already need 2^{34} bytes, i.e., 16GB. Therefore, in this chapter, we only use this table lookup algorithm for $GF(2^8)$.

Log and Antilog Tables

Since all non-zero elements in a finite field form a cyclic group under multiplication [16], there exists a primitive element α in the field so that any non-zero element in the field is a power of the primitive element: for any $g \in GF(2^n)$, there exists an $0 \leq \ell < 2^n - 1$ such that $g = \alpha^\ell$. ℓ is called *the discrete logarithm* of element g with respect to α in field $GF(2^n)$.

Based on this observation, a table lookup algorithm can be constructed [87]. This algorithm builds two tables called log and antilog. The log table records the mapping from an element g to its discrete logarithm ℓ. Conversely, the antilog table records the mapping from power ℓ to a unique element g in the field. These two tables can be built with the binary polynomial method for implementing exponentiation. After pre-computing these two tables, field operations can be performed as follows:

Multiplication of two elements g_1 and g_2: If g_1 or g_2 is 0, multiplication returns 0. Otherwise, do a lookup in the log table and get the discrete logarithms ℓ_1 and ℓ_2 for g_1 and g_2, respectively. Then, compute $\ell_3 = (\ell_1 + \ell_2) \bmod (2^n - 1)$. Finally, use the antilog table to find the field element g_3 corresponding to power ℓ_3 and return g_3 as the multiplication result.

Division of two elements g_1 and g_2: If g_1 is 0, division returns result 0. Otherwise, use the log table to lookup the discrete logarithms ℓ_1 and ℓ_2 for g_1 and g_2, respectively. Then, compute $\ell_3 = (\ell_1 - \ell_2) \bmod (2^n - 1)$. Finally, use the antilog table to find the field element g_3 corresponding to power ℓ_3 and return g_3 as the division result.

Both multiplication and division involve four similar steps: (1) determine whether one element is 0; (2) perform a lookup in the log table; (3) compute the modular addition or subtraction; (4) use the antilog table

to lookup the final result. Steps (1) and (3) could be optimized. Jerasure, for example, expands the storage of antilog tables by a factor of three to avoid step (3), which results in improved performance [93]. Huang et al. expand the antilog table by a factor of four to be able to remove both steps (1) and (3), and improve computation performance by up to 80% [56].

In this chapter, we make use of the optimizations in [56] to implement this algorithm. The time complexity for both multiplication and division is then one addition (or subtraction) operation with three table lookups. The space complexity is $5 \cdot (n/8) \cdot 2^{n+1}$ bytes. Hence, this algorithm is applicable only to $GF(2^8)$ and $GF(2^{16})$ before memory demands become unreasonable.

5.3.3 Hybrid of Computational and Table Lookup Methods

The binary polynomial method evaluates the result of an arithmetic operation each time it is invoked. On the other hand, table lookup methods pre-compute and store all the results of arithmetic operations, resulting in very fast response time when an operation is invoked. In this section, we explore hybrid approaches that combine ideas from both methods to achieve computation efficiency for large finite fields with reasonable memory consumption.

Split Tables

The split table algorithm has been proposed by Huang and implemented in Jerasure by Plank [88]. This algorithm is designed to optimize multiplication. To perform multiplication of two elements g_1 and g_2, this algorithm breaks each n-bit element in the field into $n/8$ units of size one byte. Then, it computes the result of multiplication by combining multiplication results of all unit pairs containing one byte from each operand. An example is shown below.

Multiplication of two elements g_1 and g_2: Suppose, for simplicity, that g_1 and g_2 are in $GF(2^{16})$. We represent g_1 as $[a_1, a_0]$, where a_1 is the high-order byte of g_1 and a_0 is the low-order byte of g_1. Similarly, we represent g_2 as $[b_1, b_0]$. By the distributive property of multiplication over finite fields, we can write:

$$\begin{aligned} g_1 * g_2 &= [a_1, a_0] * [b_1, b_0] \\ &= [a_1, 0] * [b_1, 0] + [a_1, 0] * [0, b_0] \\ &+ [0, a_0] * [b_1, 0] + [0, a_0] * [0, b_0] \end{aligned} \tag{5.2}$$

To perform the above multiplication efficiently, we can first use the binary polynomial method to build three multiplication tables called split tables [88]. The tables store the multiplication results of all pairs of the form $[a_1, 0] * [b_1, 0]$, $[a_1, 0] * [0, b_0]$, and $[0, a_0] * [0, b_0]$. To evaluate $g_1 * g_2$, the results of multiplication for the four pairs in Equation (5.2) are looked up in split tables, and combined by bitwise XORs.

Division: This algorithm proceeds as in the binary polynomial method, which uses the Extended Euclidean Algorithm or its variants.

In general, for $GF(2^n)$, one multiplication needs $(n/8)^2$ table lookups. In terms of the space complexity, we need to build $n/4 - 1$ split tables for $GF(2^n)$, and the size of each table is $(n/8) * 2^{16}$ bytes. Thus, the total amount of storage needed is $(n/4 - 1) * (n/8) * 2^{16} = 2n(n - 4)$ KB. For $GF(2^{64})$, this results in 7.5MB storage, an acceptable memory requirement. Therefore, this algorithm can be considered for large finite fields.

Extension Field Method

A more scalable algorithm to support large finite fields is the extension field method. This method makes use of precomputed tables in a smaller finite field, and several techniques for extending small field arithmetic into larger field operations.

Extension field theory. Section 5.3.1 describes the standard basis representation for elements of finite field $GF(2^n)$. In general, a finite field can use any of its proper base fields to represent its elements [16, 71]. For example, if $n = k \cdot m$, then field $GF(2^n)$ is isomorphic to $GF((2^k)^m)$. An element in $GF(2^n)$ can be represented as a polynomial $a_{m-1}x^{m-1} + a_{m-2}x^{m-2} \cdots + a_1x + a_0$ of degree $m - 1$ with coefficients in $GF(2^k)$. We can use an irreducible polynomial of degree m over $GF(2^k)$ to define the field arithmetic for $GF(2^n)$. With this representation, $GF(2^k)$ is named a *base field* of $GF(2^n)$, and $GF(2^n)$ an *extension field* of $GF(2^k)$.

For clarity, let us give an example for $GF(2^{16})$. If we consider it an extension field of $GF(2^8)$, then it becomes isomorphic to $GF((2^8)^2)$. We need to find two irreducible polynomials: one for the arithmetic in the base field $GF(2^8)$ (for instance $f(x) = x^8 + x^4 + x^3 + x^2 + 1$), and the second for generating the extension field $GF((2^8)^2)$ from base field $GF(2^8)$ (for instance $p(x) = x^2 + x + 32$).

Multiplication of two elements g_1 and g_2: Suppose that $g_1 = (a_1, a_0)$ and $g_2 = (b_1, b_0)$ are two elements

in $GF((2^8)^2)$, with a_0, a_1, b_0 and b_1 in $GF(2^8)$, and $p(x)$ is an irreducible polynomial of degree 2 over $GF(2^8)$. Multiplication of g_1 and g_2 is performed as follows:

$$\begin{aligned} & (a_1x + a_0) * (b_1x + b_0) \\ = \; & (a_1 * b_1)x^2 + (a_1 * b_0 + a_0 * b_1)x + a_0 * b_0 \bmod p(x) \\ = \; & (a_1 * b_0 + a_0 * b_1 + 32 * a_1 * b_1)x \\ + \; & (a_0 * b_0 + 32 * a_1 * b_1) \end{aligned}$$

As all coefficients of g_1 and g_2 are from $GF(2^8)$, the multiplications and additions of coefficients in the above computation are performed in base field $GF(2^8)$. Addition is implemented as bitwise XOR, and multiplication in $GF(2^8)$ as table lookup.

For a general $GF(2^n)$, the time complexity of multiplication depends on the base field and the irreducible polynomial $p(x)$. One multiplication in the extension field $GF((2^k)^m)$ needs at least m^2 multiplications in the base field $GF(2^k)$. Let us give a justification for this theoretical lower bound. There are two steps involved in the multiplication of two elements in the extension field: multiplication of two polynomials of degree $m-1$ (resulting in m^2 multiplications in the base field), and reduction modulo the irreducible polynomial generating the extension field. If the irreducible polynomial used for generating the extension field has coefficients of only 0 or 1, no additional multiplications are needed in the second step. In practice, this bound may not be reachable since such an irreducible polynomial may not exist for some combinations of $GF(2^k)$ and m. More discussion on how to choose an irreducible polynomial $p(x)$ that reduces the number of multiplications in the base field is given in Section 5.4.

The space complexity for multiplication in extension field $GF(2^n)$ with $n = k \cdot m$ is exactly the same as that of base field $GF(2^k)$, and is thus independent from the extension field.

Division of two elements g_1 and g_2: Division in the extension field contains two steps: finding the inverse of g_2, and multiplying g_1 with g_2^{-1}. Computing the inverse of an element in the extension field can be implemented with the Extended Euclidean Algorithm. The Binary Extended Euclidean Algorithm and the Almost Inverse Algorithm used in the binary polynomial method are not applicable for extension fields.

5.4 Efficient Implementation of Operations in Extension Fields

In this section, we describe the main contribution of the chapter, consisting of efficient implementation techniques for the extension field method.

5.4.1 Irreducible Polynomials

When implementing the extension field method, one important factor that impacts the performance of arithmetic operations is the choice of the irreducible polynomial used to construct the extension field. The irreducible polynomial determines the complexity of polynomial modular reduction and hence greatly affects multiplication and division performance. In general, there are multiple choices for irreducible polynomials, and our goal is to find those that optimize the performance of arithmetic operations.

Impact of Irreducible Polynomials

We give an example to demonstrate the great impact of irreducible polynomials on multiplication performance. Consider the extension field $GF((2^8)^4)$. When using $f(x) = x^8 + x^4 + x^3 + x^2 + 1$ as the irreducible polynomial over $GF(2)$ for $GF(2^8)$, there are two irreducible polynomials of degree 4 over $GF(2^8)$: $p_1(x) = x^4 + x^2 + 6x + 1$ and $p_2(x) = x^4 + 2x^2 + 5x + 3$. Either of them can be used to construct $GF((2^8)^4)$. The multiplication complexity in $GF((2^8)^4)$, however, is significantly different for these two irreducible polynomials and is shown in Table 5.1.

Irreducible polynomial	Multiplication	Addition
$x^4 + x^2 + 6x + 1$	16+3	18
$x^4 + 2x^2 + 5x + 3$	16+9	18

Table 5.1: Multiplication complexity in $GF((2^8)^4)$ when using two different irreducible polynomials.

In Table 5.1, the second column shows the number of multiplications in the base field, and the third shows the number of additions in the base field. As multiplication is a much slower operation than addition, the number of multiplications in the base field dominates the performance.

In the second column, each number consists of two terms: the first term is the number of multiplications in the base field when multiplying two polynomials, and the second term is the number of multiplications performed when reducing the multiplication result modulo the irreducible polynomial. In our example, when multiplying two polynomials in $GF((2^8)^4)$, the cost of the first term is fixed, i.e., $4^2 = 16$ multiplications in

$GF(2^8)$ (in this chapter, we exclude the low probability of having the same multiplication pairs in the base field when performing a single multiplication for the extension field.) This number is independent of the irreducible polynomial $p(x)$ we are using. The cost of the second term, however, is determined by $p(x)$, and it can vary dramatically. In Table 5.1, this cost is 3 multiplications for $p_1(x)$, but 9 multiplications for $p_2(x)$. Hence, the multiplication complexity for using $p_1(x)$ is 19 multiplications compared to 25 multiplications for $p_2(x)$, resulting in a 24% improvement in performance.

For larger finite fields, such as $GF(2^{128})$, the difference in performance between using a carefully chosen irreducible polynomial and a random one would be even more significant. Therefore, it is important to find efficient irreducible polynomials for optimizing performance.

Test of Irreducible Polynomials

There are many efficient irreducible polynomials over base field $GF(2)$ listed in the literature [102]. However, for an arbitrary field, we have to search for good irreducible polynomials. During the search process, one key step is testing whether a polynomial is irreducible or not. A fast test algorithm is the Ben-Or algorithm [17, 38]. With the Ben-Or algorithm, we developed a test program using the NTL library [108]. Our experience shows that combing the Ben-Or algorithm with NTL leads to an efficient algorithm for testing polynomial irreducibility.

Efficient Irreducible Polynomial Patterns

Section 5.4.1 shows that the choice of irreducible polynomial greatly affects modular reduction efficiency. Particularly, it determines the number of multiplications in the base field. There is one key parameter of the irreducible polynomial that decides the number of multiplications performed in the base field: the number of coefficients not in $GF(2)$ (i.e., the number of coefficients different from 0 and 1, the only elements in $GF(2)$). We develop heuristics to search for irreducible polynomials that have the least number of coefficients not in $GF(2)$. In addition, we try to reduce the number of coefficients of 1 to decrease the number of additions during modular reduction.

We present some efficient irreducible polynomial patterns that we found through our search heuristics in Table 5.2.

The 1st column of Table 5.2 is the value of n in extension field $GF(2^n)$. The 2nd column is irreducible polynomials over base field $GF(2^8)$ for $GF(2^n)$, and the 3rd column is over base field $GF(2^{16})$. The irreducible polynomial used to construct $GF(2^8)$ is $f(x) = x^8 + x^4 + x^3 + x^2 + 1$ over $GF(2)$, and it is

n	$GF(2^8)$	$GF(2^{16})$
32	$x^4 + x^2 + 6x + 1$	$x^2 + x + 8192$
48	$x^6 + x^2 + x + 32$	$x^3 + x + 1$
64	$x^8 + x^3 + x + 9$	$x^4 + x^2 + 2x + 1$
80	$x^{10} + x^3 + x + 32$	$x^5 + x^2 + 1$
96	$x^{12} + x^3 + x + 2$	$x^6 + x^3 + 8192$
112	$x^{14} + x^3 + x + 33$	$x^7 + x + 1$
128	$x^{16} + x^3 + x + 6$	$x^8 + x^3 + x + 8$

Table 5.2: Irreducible polynomials for extension fields $GF(2^n)$ over base field $GF(2^8)$ and $GF(2^{16})$

$f(x) = x^{16} + x^{12} + x^3 + x + 1$ for $GF(2^{16})$.

It can be proved that above irreducible polynomials are optimal in terms of the number of coefficients not in $GF(2)$. We consider two cases. 1) When constructing $GF(2^k)^m$, k and m are relative prime. For such k and m, the presented irreducible polynomials in Table 5.2 only contain coefficients in $GF(2)$, so they are optimal. 2) k and m are not relative prime. A fact is that if a polynomial of degree m that is irreducible over $GF(2)$, is also irreducible over $GF(2^k)$, then $\texttt{gcd}(m, k)$=1 [71, ch. 3.3]. Thus, when k and m are not relative prime, equation $\texttt{gcd}(m, k)$=1 does not hold, and then any irreducible polynomial with degree m over $GF(2^k)$ must have at least one coefficient not in $GF(2)$. In this case, because the presented irreducible polynomials in Table 5.2 contain only one coefficient not in $GF(2)$, they are also optimal.

With the above irreducible polynomials, one multiplication in $GF((2^k)^m)$ can be performed with m^2 or $m^2 + m - 1$ multiplications in base field $GF(2^k)$. If k and m are relative prime, the multiplication number is m^2; otherwise, it is $m^2 + m - 1$. As explained in Section 5.3.3, there are two steps involved in the multiplication of two elements in the extension field. The first is the multiplication of two polynomials of degree $m - 1$ (resulting in m^2 multiplications in the base field), and the second is the modular reduction modulo the irreducible polynomial generating the extension field. If the irreducible polynomial only contains coefficients in $GF(2)$, the second step needs 0 multiplication; otherwise, if there is one coefficient not in $GF(2)$, the second step results in $m - 1$ multiplications.

5.4.2 Multiplication Implementation

This section presents the multiplication implementation for the extension field method. For simplicity, we focus on the implementation for extension fields $GF((2^k)^m)$ where $\texttt{gcd}(m, k) \neq 1$. The implementation for the simpler case where $\texttt{gcd}(m, k)$=1 can be easily derived.

In Section 5.4.1, we gave an example showing how the choice of the irreducible polynomial generating

an extension field affects the efficiency of multiplication. Table 5.1 shows that with polynomial $p_1(x)$, we need to perform 19 multiplications in the base field $GF(2^8)$ for each multiplication in the extension field $GF((2^8)^4)$. If multiplication in the base field is implemented with full multiplication and division tables, this corresponds to 19 table lookups in the base field $GF(2^8)$ as one multiplication needs only one table lookup.

However, if log and antilog tables are used for multiplication in the base field, the number of lookups increases by a factor of three. This is because one multiplication now involves three table lookups, two to the log table and one to the antilog table. In this section, we provide an efficient multiplication algorithm for using log and antilog tables in the base field. The implementation greatly decreases the number of table lookups from $3(m^2 + m - 1)$ to $m^2 + 4m - 1$ for fields of the form $GF((2^8)^m)$ and $GF((2^{16})^m)$. This is achieved by caching table lookup results and using them repeatedly.

The implementation contains two algorithms: the multiplication algorithm using log and antilog tables and the modular reduction algorithm specifically designed for the irreducible polynomials presented in Section 5.4.1. In the multiplication algorithm given in Algorithm 9, we multiply two polynomials $a(x)$ and $b(x)$ of degree at most $m - 1$ with coefficients in the base field and output as a result a polynomial $c(x)$ of degree at most $2m - 2$.

In Algorithm 9, a, b, and c are coefficient vectors of polynomials $a(x)$, $b(x)$, and $c(x)$, respectively. Each element of these vectors represents a single coefficient in the base field. The variables *logtable* and *antilogtable* are lookup tables in the base field. They are built in advance.

Algorithm 9 Multiplication using log and antilog tables

INPUT: Polynomials $a(x)$ and $b(x)$ of degree at most $m - 1$
OUTPUT: Polynomial $c(x) = a(x) \cdot b(x)$ of degree at most $2m - 2$

```
1:  c ← 0;
2:  for k from 0 to m − 1 do
3:      alog[k] = logtable[a[k]];
4:      blog[k] = logtable[b[k]];
5:  end for
6:  for k1 from 0 to m − 1 do
7:      for k2 from 0 to m − 1 do
8:          c[k1 + k2] ⊕= antilogtable[alog[k1] + blog[k2]];
9:      end for
10: end for
```

As the output $c(x)$ of Algorithm 9 may have degree more than m, it has to be modularly reduced. Here, we provide an efficient modular reduction algorithm. Suppose the irreducible polynomial is of the form

$p(x) = x^m + x^3 + x + v$. The powers $x^{2m-2}, \ldots, x^m$ can be reduced modulo $p(x)$ as follows:

$$\begin{aligned} x^{2m-2} &\equiv (x^3 + x + v) \cdot x^{m-2} \bmod p(x) \\ x^{2m-3} &\equiv (x^3 + x + v) \cdot x^{m-3} \bmod p(x) \\ x^{2m-4} &\equiv (x^3 + x + v) \cdot x^{m-4} \bmod p(x) \\ &\ldots \\ x^{m+1} &\equiv (x^3 + x + v) \cdot x \bmod p(x) \\ x^m &\equiv x^3 + x + v \bmod p(x) \end{aligned}$$

Our developed reduction method is presented in Algorithm 10, which is similar to the modular reduction approach in [113]. In Algorithm 10, c and d are coefficient vectors of input polynomial $c(x)$ of degree $2m-2$ and output polynomial $d(x)$ of degree $m-1$, respectively. Similarly efficient algorithms could be given for other patterns of $p(x)$. Note that this algorithm reduces the degree of $c(x)$ by one each time the loop is executed (lines 2-6).

Algorithm 10 Modular reduction

INPUT: Polynomial $c(x)$ of degree at most $2m-2$
OUTPUT: Polynomial $d(x) = c(x)$ mod $p(x)$ of degree at most $m-1$

1: $vlog$ = logtable[v];
2: **for** k from $2m-2$ to m **do**
3: $c[k-(m-3)] \oplus = c[k]$;
4: $c[k-(m-1)] \oplus = c[k]$;
5: $c[k-m] \oplus =$ *antilogtable*[*logtable*[$c[k]$] + $vlog$];
6: **end for**
7: $d \leftarrow 0$;
8: **for** k from 0 to $m-1$ **do**
9: $d[k] = c[k]$;
10: **end for**

We proceed to analyze the complexity of our multiplication method. In Algorithm 9, $2m$ table lookups are performed in lines 2-5, and m^2 table lookups are performed in lines 6-10. In Algorithm 10, $2m-1$ table lookups are performed in lines 1-6. Adding all operations in Algorithms 9 and 10, we obtain the multiplication complexity: $m^2 + 4m - 1$ table lookups.

Similar multiplication and modular reduction algorithms can be derived by using full multiplication and division tables to implement operations in the base field. The corresponding time complexity is $m^2 + m - 1$ table lookups.

5.5 Performance Evaluation

In this section we evaluate the algorithms described above for large finite fields ranging from $GF(2^{32})$ to $GF(2^{128})$. We present performance results for different multiplication and division algorithms within a field. We evaluate its performance improvement from the use of our newly implemented algorithms compared to previous implementations.

5.5.1 Experiment Setup

Platforms

The multiplication and division tests were run on a variety of platforms in order to observe how their performances vary on different processors. We tested our implementations on four platforms, all using Intel 64-bit processors, spanning their current offering from low to high-end. Table 5.3 details the specifications of each platform.

Platform	CPU speed	L2 cache	Model
P4	3.0GHz	2MB	Pentium 4
Pd	2.8GHz	$2 \cdot 1$MB	Dual Core (D820)
Pc2d	2.1GHz	3MB	Core 2 Duo (T8100)
Pc2q	2.4GHz	$2 \cdot 4$MB	Core 2 Quad (Q6600)

Table 5.3: Platforms under test.

All tests were run on a 64-bit version of Linux. As a result, one `int` value represents one element in $GF(2^{32})$; one `long` value, i.e., a computer word, holds an element in $GF(2^{48})$ and $GF(2^{64})$; two `long` values represent an element in fields from $GF(2^{80})$ up to $GF(2^{128})$.

Implementations

We evaluated five implementations listed in Table 5.4, representing three distinct methods. Throughout the rest of the chapter, we simply use the names in the first column of Table 5.4 to refer to various implementations. *binary* is based on the binary polynomial method from Section 5.3.1; specifically, it uses the left-to-right comb method with windows of width w ($w = 4$) [73] for multiplication and the Binary Extended Euclidean Algorithm [81] for division. Similar to [73, 10, 8], we chose width $w = 4$, which we expect provides optimum performance. *split* uses the split table method for multiplication from Section 5.3.3 and the same division algorithm as *binary*. *gf8 (full)*, *gf8 (log)* and *gf16 (log)* are based on the extension field method

from Section 5.3.3. *gf8 (full)* uses base field $GF(2^8)$, with arithmetic operations based on full multiplication and division tables. *gf8 (log)* also uses base field $GF(2^8)$, but it implements arithmetic operations in $GF(2^8)$ by log and antilog tables. *gf16 (log)* is based on $GF(2^{16})$ with operations implemented using log and antilog tables.

Implementation	Method
binary	binary polynomial
split	split table
gf8 (full)	extension field
gf8 (log)	extension field
gf16 (log)	extension field

Table 5.4: Evaluated implementations for $GF(2^n)$.

We developed all implementations for finite fields of interest, and borrowed the implementations of arithmetic operations in $GF(2^8)$ and $GF(2^{16})$ from Jerasure. The code is written in C and compiled using **gcc** with the **-O2** optimization flag, which is recommended for most applications [39]. The code is single-threaded, and thus does not take advantage of multiple cores when present.

In addition to compiler optimizations, many manual optimizations are applied to each individual implementation for best performance. One common optimization is that we do not use one general multiplication or division function for all tested finite fields, but instead develop specific implementations for each field. This allows us to determine the size of data structures at compile time, rather than runtime, which improves performance significantly. Another two important optimizations are performed in the implementation of the left-to-right comb method with windows of width w ($w = 4$) for implementation *binary*. First, in Algorithm 8, we manually unroll the loop from line 6 to line 8. Second, line 11 is actually an iteration. We also manually unroll this loop. We found that these two optimizations greatly improve multiplication performance. For instance, we achieve an improvement of 20% for $GF(2^{96})$ and 35% for $GF(2^{128})$ on platform Pc2q.

5.5.2 Comparison of All Implementations Using Table Lookups

This section compares the performance of all implementations heavily using table lookups. Regarding these implementations, table lookup is a dominant performance factor, so we first present table lookup numbers of each implementation in Table 5.5. The table lists the number of table lookups needed for one multiplication (column 2). Table 5.5 shows that *gf16 (log)* performs the least number of table lookup operations and

is followed by *split*. *gf8 (full)* and *gf8 (log)* need more table lookups than the previous two. It is worth noting that for implementation *gf16 (log)*, if $\texttt{gcd}(n/16, 16) \neq 1$, the table lookup number is $n^2/256 + 4n/16 - 1$; otherwise, it is $n^2/256 + 2n/16$.

The space complexity of each implementation is given in the 3rd column of Table 5.5. The size listed here is the combined space needed for both the multiplication and division algorithms. Note that the implementations based on the extension field method (*gf8 (full)*, *gf8 (log)*, and *gf16 (log)*) all use one `int` value to represent an element in $GF(2^8)$ or $GF(2^{16})$, which over-estimates the minimum space requirement, but leads to faster arithmetic operations. The table shows that the memory requirements for implementations based on the extension field method are independent of the size of the finite field. However, *split* consumes memory quadratic in n, limiting its practicality for large finite fields.

Implementation	Table lookups	Memory needed
split	$n^2/64$	$2n(n-4)$ KB
gf8 (full)	$n^2/64 + n/8 - 1$	0.5 MB
gf8 (log)	$n^2/64 + 4n/8 - 1$	5 KB
gf16 (log)	$n^2/256 + 4n/16 - 1$ (or $n^2/256 + 2n/16$)	1.25 MB

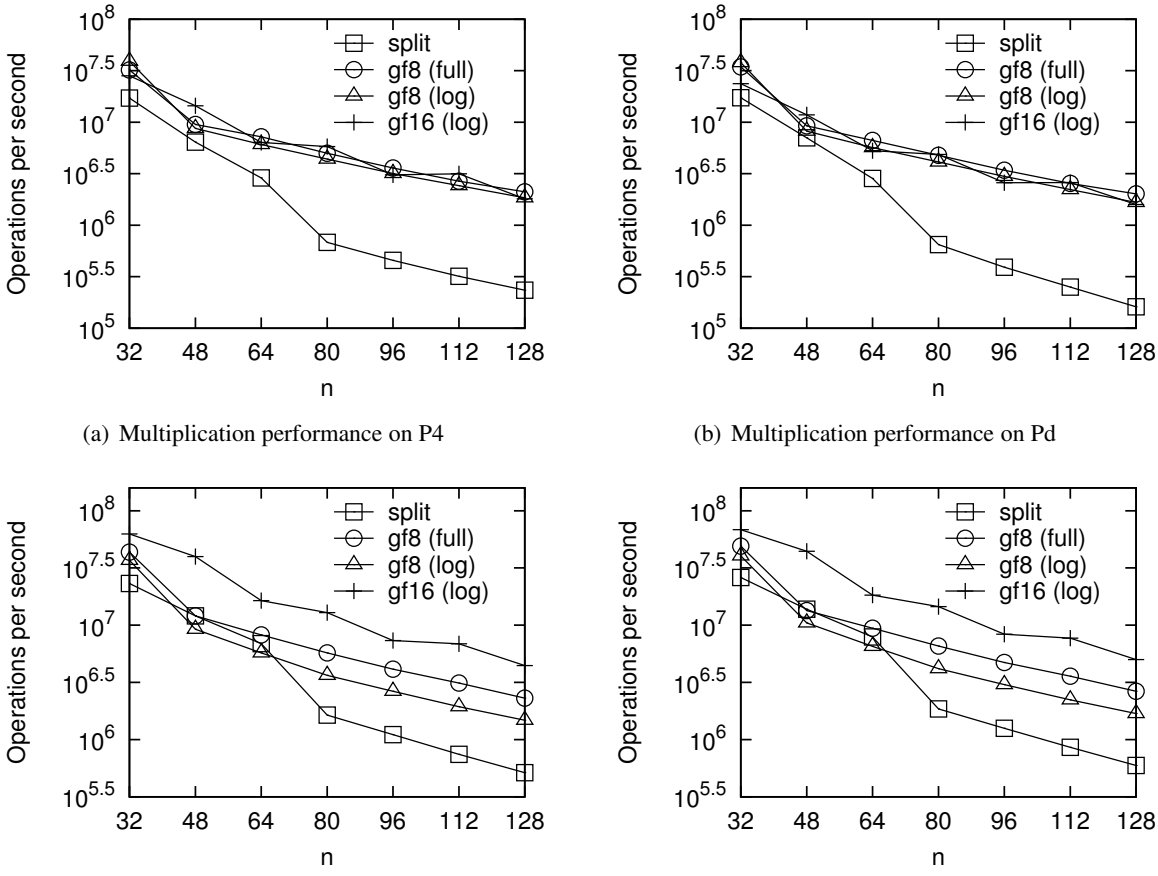
Table 5.5: Table lookup number and memory needed of various implementations.

We now compare the measured performance of all these implementations. To measure the raw performance of an implementation, we randomly pick 36,000,000 element pairs from a finite field. The values of all element pairs are generated randomly. We then either multiply the pair, or divide one element by the other, and measure the elapsed time through the use of the `gettimeofday()` system call. Finally, we calculated how many operations are performed per second (operations per second). Each experiment is run 30 times and the average result is plotted in the graphs that follow. When the confidence level is set at 95%, the margin of error to the average value is less than 5% for all data points. As the margin of error is so small, we do not display error bars in the following figures.

Multiplication

Figure 5.1 displays the absolute multiplication performance on all four platforms. In the figure, the X-axis is the value of n. The Y-axis is the number of multiplications performed in one second, in base-10 log scale. Below are some observations that can be drawn from the data.

1. Among table lookup intensive implementations, *gf16 (log)* outperforms all other implementations in most cases. The reason is that *gf16 (log)* performs about a quarter of the table lookups compared to



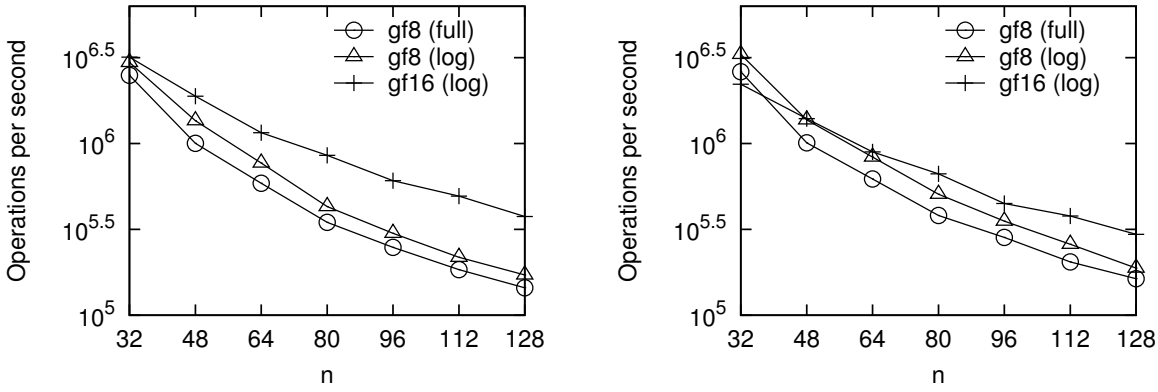
(a) Multiplication performance on P4

(b) Multiplication performance on Pd

(c) Multiplication performance on Pc2d

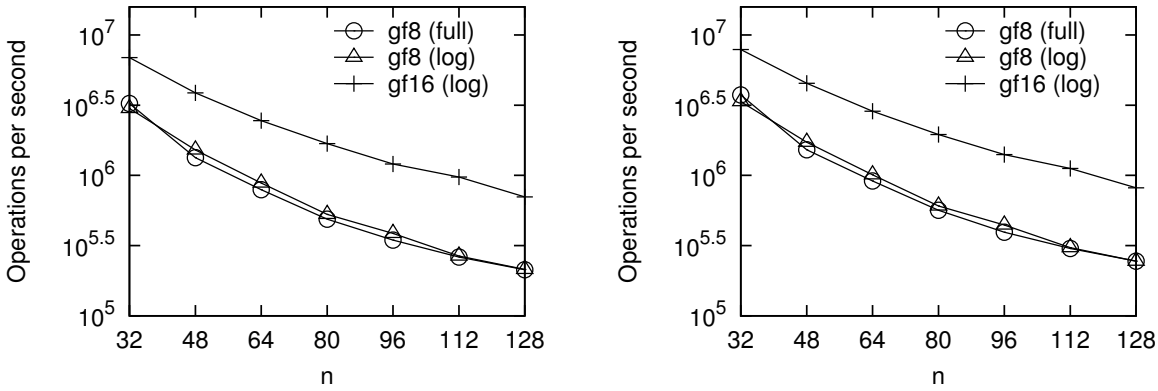
(d) Multiplication performance on Pc2q

Figure 5.1: Multiplication performance of $GF(2^n)$ on various platforms.



(a) Division performance on P4

(b) Division performance on Pd



(c) Division performance on Pc2d

(d) Division performance on Pc2q

Figure 5.2: Division performance of $GF(2^n)$ on various platforms.

other implementations, as shown in Table 5.5. The performance gap is not significant in platform P4 and Pd due to their small L2 caches.

2. *gf8 (full)* performs better than *gf8 (log)*, but the difference depends on platforms. *gf8 (full)* needs less table lookups than *gf8 (log)*, but its greater memory needed result in higher CPU cache miss ratio on platforms with small CPU cache, and thus these two implementations achieve similar performance on platforms P4 and Pd. However, on other two platform Pc2d and Pc2q, *gf8 (full)* outperforms *gf8 (log)* due to their large CPU caches.

3. *split* has the worst performance of all the implementations in most cases. Although it uses a similar number of table lookups as *gf8 (full)* and *gf8 (log)*, it uses memory quadratic in n. This causes a large number of cache misses in larger fields, resulting in much worse performance.

4. As the size of our finite fields grows, the absolute performance of all table lookup intensive implementations decreases due to the increasing number of table lookups. All these implementations heavily depend on table lookups, and thus, their performance degrades almost linearly as the number of table lookups increases.

Division

Figure 5.2 displays the absolute division performance for all implementations except *split*. We omit *split* here as it uses the same division algorithm as *binary*. We observe the following:

1. *gf16 (log)* performs best among table lookup intensive implementations. As with multiplication, the performance advantage is highlighted on platforms with larger L2 caches.

2. *gf8 (full)* performs worse than *gf8 (log)* on platforms with small L2 caches, but they perform similarly on platforms with large L2 caches.

3. As with multiplication, as finite field size increases, the division performance of table lookup intensive implementations also decreases. However, the rate of decrease in division is slower than that of multiplication. For example, in platform Pc2q, the division performance of *gf16 (log)* on finite field $GF(2^{32})$ is 7% of its performance on finite field $GF(2^{128})$, while this number of multiplication performance is 10%.

Summary

The above results show that, among table lookup intensive implementations, *gf16 (log)* performs best and in most cases, by a large margin. This observation is consistent with the analysis shown in Table 5.5, which indicates that *gf16 (log)* performs about a quarter of the table lookups and bitwise operations compared to other multiplication implementations. The memory requirements for *split* cause it to perform the worst, while the performances of *gf8 (full)* and *gf8 (log)* are between that of *split* and *gf16 (log)*.

5.5.3 Comparison of *binary* and *gf16 (log)*

This section focuses on comparing the multiplication and division performance of *binary* and *gf16 (log)*, the best one among table lookup implementations.

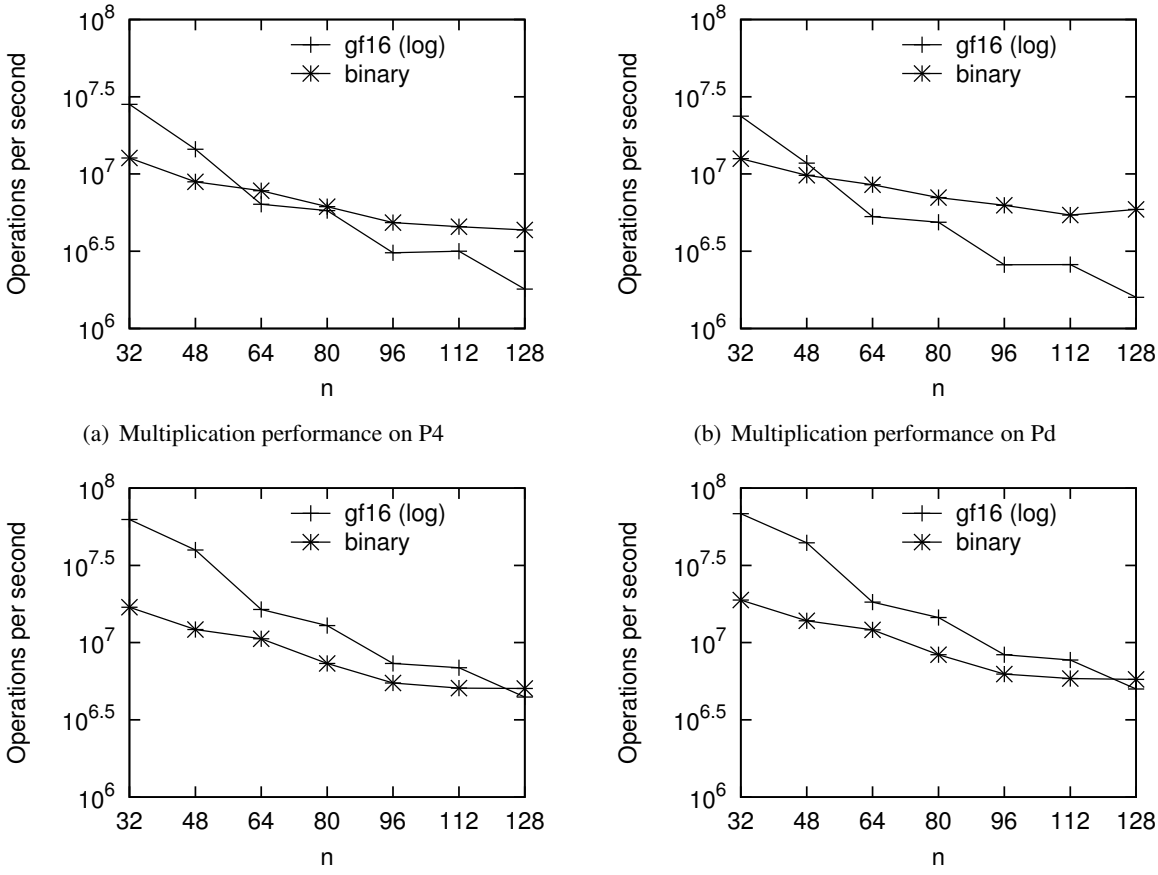
Multiplication

In Figure 5.3, graphs 5.3(a) - 5.3(d) display the multiplication performance comparison of *binary* and *gf16 (log)*. We have the following observations:

1. *gf16 (log)* performs better than *binary* from $GF(2^{32})$ to $GF(2^{48})$ on platforms P4 and Pd which only have at most 2MB L2 caches; on platforms Pc2d and Pc2q that have at least a 3MB L2 cache, *gf16 (log)* performs better from $GF(2^{32})$ to $GF(2^{112})$. For example, on Pc2q, the performance of *gf16 (log)* is higher than that of *binary* by 200% for field $GF(2^{32})$ and 35% for field $GF(2^{112})$.

2. *gf16 (log)* outperforms *binary* for finite field $GF(2^{32})$ in all platforms, but as the finite field size grows, the performance gap between *gf16 (log)* and *binary* gradually decreases, with *binary* eventually outperforming *gf16 (log)*. This transition occurs in all graphs. We could explain this effect using the analysis in Table 5.5. The performance of *gf16 (log)* is dominated by table lookups, which is quadratic in the value of n, and thus its performance greatly decreases when n grows. The *binary* implementation, however, is a computation intensive implementation and thus degrades less significantly.

3. *gf16 (log)* starts performing slower than *binary* from finite field $GF(2^{64})$ on platforms P4 and Pd, while the starting field is $GF(2^{128})$ on platforms Pc2d and Pc2q. This is because P2d and Pc2q contains much larger CPU caches than P4 and Pd, and large cache size improves the performance of *gf16 (log)*, a table lookup intensive implementation. As L2 caches are currently increasing in size faster than CPU speed, *gf16 (log)* has the potential to surpass *binary*, for larger finite fields, in the near



(a) Multiplication performance on P4

(b) Multiplication performance on Pd

(c) Multiplication performance on Pc2d

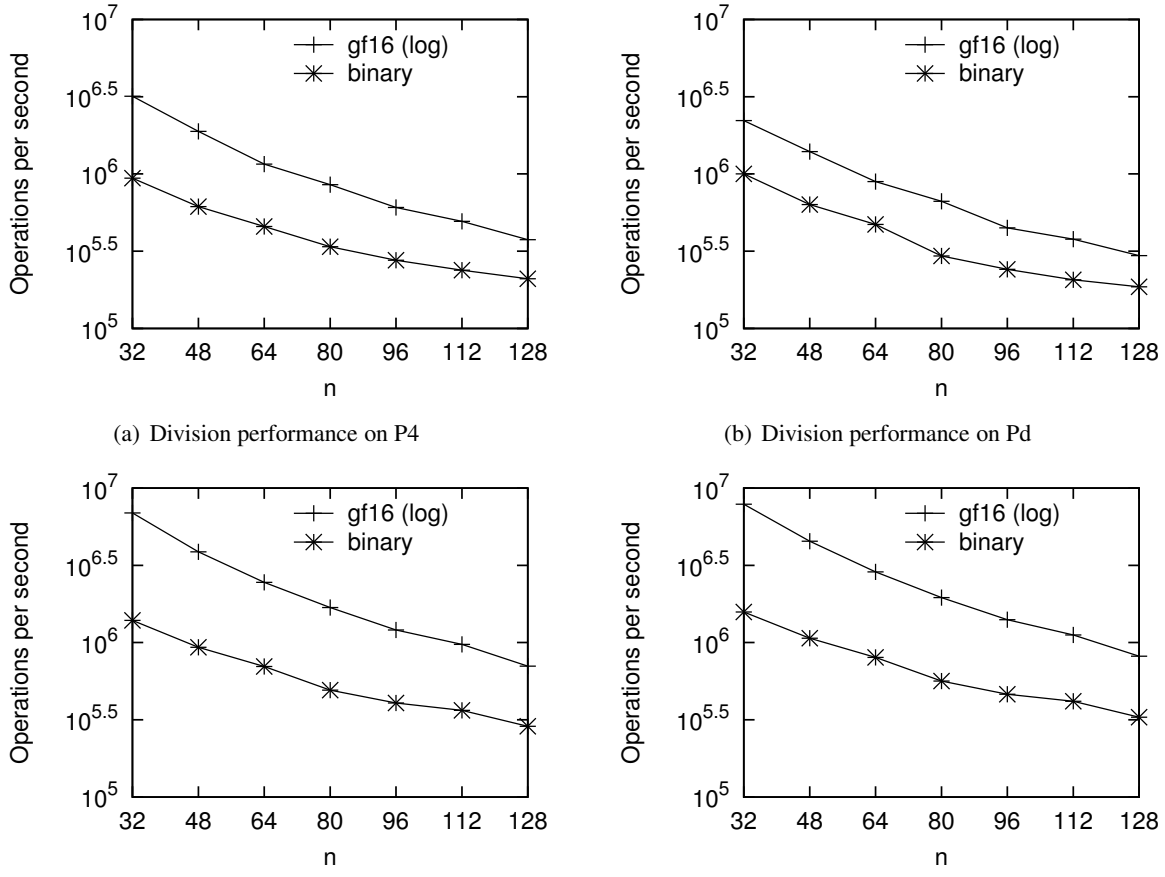
(d) Multiplication performance on Pc2q

Figure 5.3: Multiplication performance of *binary* and *gf16 (log)* on various platforms.

future. Nonetheless, the performance trend of *binary* makes it the best choice for extremely large finite fields for the foreseeable future.

Division

Graphs 5.4(a) - 5.4(d) display the division performance comparison. The figure shows that in all cases, *gf16 (log)* greatly outperforms *binary*. For example, on platform Pc2q, the performance of *gf16 (log)* is 100% higher than that of *binary* for field $GF(2^{128})$ and 300% for field $GF(2^{32})$. Similar performance improvement can be observed on all other platforms. The reason is that the division algorithm of *binary* i.e., Binary Extended Euclidean Algorithm, works on one bit at a time since its base field is $GF(2)$. The division algorithm of *gf16 (log)*, however, works on 16 bits at a time due to its base field being $GF(2^{16})$. Hence, *gf16 (log)* is much more efficient than *binary* on division.



(a) Division performance on P4

(b) Division performance on Pd

(c) Division performance on Pc2d

(d) Division performance on Pc2q

Figure 5.4: Division performance of *binary* and *gf16 (log)* on various platforms.

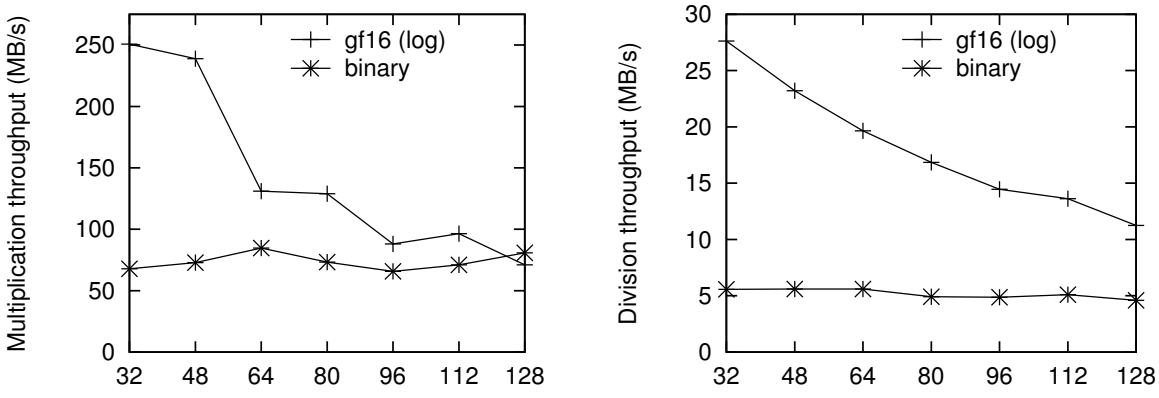
Throughput Comparison

To better understand the performance of finite fields from application perspective, we compared the performance of *gf16 (log)* with *binary* by throughput, i.e., how much data can be processed per second. As throughput is visible to applications, the performance comparison would be more useful in practice. The multiplication/division throughput is calculated as:

$$\text{Throughput} = \frac{\text{Operations per second * n}}{8} \tag{5.3}$$

This is derived as follows. Let O be Operations per second. Then, On is how many bits processed per second for multiplication or division, and $\frac{On}{8}$ is bytes processed per second, or Equation 5.3 for throughput.

The throughput comparison is presented in Figure 5.5. Here, we only show comparison results on platform Pc2d. Interested readers can easily do the same conversions for other platforms from Equation 5.3. As for each finite field $GF(2^n)$, n is the same to *gf16 (log)* and *binary*, operations per second determines the throughput of the implementations. Hence, Figure 5.5(a) shows the same performance comparison results of the two implementations as Figure 5.3(c). This also applies to the division performance shown in Figure 5.5(b) and 5.4(c). However, Figure 5.5(a) and 5.5(b) display different performance trends from their counterparts. In both figures, *gf16 (log)* degrades as n grows, but *binary* keeps almost constant for all n. Because *gf16 (log)* achieves much higher performance than *binary* for $GF(2^{32})$, *gf16 (log)* can keep its performance advantage over $GF(2^{32})$ for a wide range of n values. But for multiplication, *binary* performs better when n starts from 128.



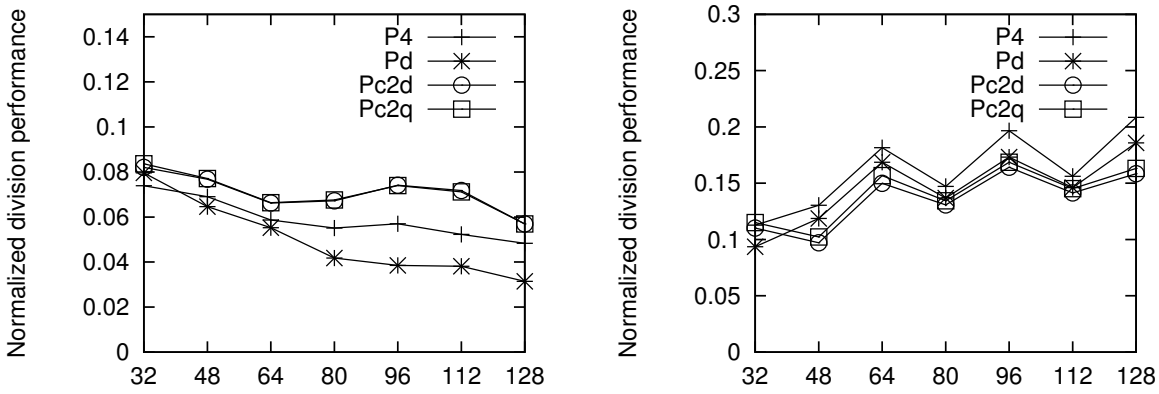
(a) Multiplication performance

(b) Division performance

Figure 5.5: Throughput comparison on platform Pc2d.

Comparison of Division and Multiplication

Because of the use of the Euclidean algorithm and its variants for implementing division, it is difficult to analyze the exact theoretical complexity of division. Here we focus on comparing the measured performance of division relative to that of multiplication. Figure 5.6 shows the normalized division performance for *binary* and *gf16 (log)*, computed as the division throughput divided by the multiplication throughput for each finite field. We make the following observations:



(a) *binary* implementation

(b) *gf16 (log)* implementation

Figure 5.6: Normalized division performance on various platforms.

1. For *binary*, the normalized division performance decreases from about 0.08 to 0.03 across all test platforms. This is because the Binary Extended Euclidean Algorithm used in *binary* contains several conditional branches and iterations, and these greatly affect the division efficiency. The left-to-right comb method with windows of width w ($w = 4$), however, has few conditional branches and only one iteration, allowing it to scale better as field size increases. This effect can be observed in Figure 5.3 too, which shows that the multiplication performance of *binary* degrades slower than that of division on all platforms.

2. For *gf16 (log)*, the normalized division performance is fairly constant at about 0.15, regardless of finite field size. As table lookup operations dominate the multiplication and division performance, this result suggests that the table lookup complexity for division is $\Theta(n^2)$.

Comparisons with Existing Implementations

We compared the multiplication performance (in operations per second) observed from our implementations with the performance reported in existing literature. The comparison results are presented in Table 5.6.

n	*gf16 (log)*	LD-lcomb(w)	n	LD-lcomb(w) [Avanzi 2007]	LD-lcomb(w) [8]
48	39,820,605	12,149,787	47	15,625,000	2,969,642
64	16,375,242	10,579,802	59	8,547,009	2,836,374
80	12,889,563	7,328,084	79	5,847,953	1,981,301
96	7,340,030	5,474,563	89	4,830,918	1,946,795
112	6,883,625	5,073,405	109	3,649,635	1,902,230
128	4,440,471	5,046,587	127	3,278,689	1,958,632

Table 5.6: Multiplication performance comparison with existing implementations for $GF(2^n)$

In Table 5.6, column 1 to column 3 shows the performance results from our experiments on platform Pc2d. The 1st column is the value of n in finite field $GF(2^n)$. The 2nd column and the 3nd column are the implementations performance of *gf16 (log)* and LD-lcomb(w), i.e., left-to-right comb method with windows of width w ($w = 4$). The 5th column is the performance of LD-lcomb(w) reported in [10] for finite fields listed in 4th column. The 6th column is our measured performance of the open source library `relic-toolkit` [8], a cryptographic library with emphasis on both efficiency and flexibility [8]. The tested field sizes for this library are in column 4.

First we compare our performance results with those in [10]. Avanzi et al. reported the performance of finite fields from $GF(2^{43})$ to $GF(2^{283})$, but we only list here their results for finite fields whose sizes are close to the ones we tested. As the performance presented in [10] is given in timing units, we translate their results to operations per second for comparison. Table 5.6 shows that our implementation of LD-lcomb(w) achieves much higher performance (column 3) than those reported in [10] (column 5) in most cases. While we employ a slightly faster platform (Core 2 Duo with 2.1 GHz CPU compared to Core 2 Duo with 1.83 GHz CPU used by Avanzi et al.), we believe that the improvement is mainly due to our optimizations.

We also compare the performance of our implemented LD-lcomb(w) with that developed in `relic-toolkit` [8]. `relic-toolkit` is a general library, which supports arithmetic operations for any finite field when its irreducible polynomial is specified. We collected the performance of `relic-toolkit` by running it on platform Pc2d. Although `relic-toolkit` provides a framework to measure performance of field operations, we did not use their framework. Instead, we used our test framework and measured the performance of their implementation of LD-lcomb(w). Column 3 of Table 5.6 shows that our implementation of LD-lcomb(w) greatly outperforms `relic-toolkit` (column 6). By looking into the source code of `relic-toolkit`, we found that `relic-toolkit` uses a general multiplication function, named `fb_mul_lodah`, to perform multiplication for all finite field sizes. The generality of this solution unavoid-

ably sacrifices performance as it is not optimized for specific field sizes. In contrast, in our implementation the multiplication operation is tailored for different size fields. This confirms that manual optimizations are still necessary to optimize performance, as observed in [10].

Again, Table 5.6 shows that the performance of *gf16 (log)* (column 2) is much higher than that of LD-lcomb(w) (column 3) in finite fields with sizes between $GF(2^{48})$ and $GF(2^{112})$.

5.6 Summary

This chapter provides new optimizations and efficient implementations of arithmetic operations for large finite fields $GF(2^n)$, ranging from $GF(2^{32})$ to $GF(2^{128})$ tailored for secure storage applications. We consider five different implementations based on three general methods for field arithmetic. We analyze the time and space complexity for these implementations, showing tradeoffs between the amount of computation and amount of memory space they employ. We also evaluate the raw performance of these implementations on four distinct hardware platforms, and present an application of our large field arithmetic implementation for distributed cloud storage.

Among the table lookup intensive implementations, we show that an implementation called *gf16 (log)*, based on the extension field method, achieves the best performance. The implementation uses precomputed log and antilog tables in $GF(2^{16})$ to speed up multiplication. We also compare the performance of *gf16 (log)* with that of the computation intensive implementation based on the binary polynomial method, called *binary*. The *binary* implementation uses the left-to-right comb method with windows of width w for multiplication and the Binary Extended Euclidean Algorithm for division. We show that in platforms with small CPU cache, multiplication in *gf16 (log)* outperforms *binary* up to finite field $GF(2^{48})$. In platforms with large CPU cache, the range extends to $GF(2^{112})$. For division, *gf16 (log)* performs best in all cases. We conclude that *gf16 (log)* is an efficient implementation for large finite fields, particularly for modern CPUs with large CPU caches.

CHAPTER 6 HyFS: A Highly Reliable File System

6.1 Introduction

Data reliability is a crucial issue to any commercial or scientific applications, which heavily depend on accessing data constantly to run businesses or conduct researches. In a computer system, data is stored on data storage systems, and then data reliability is essentially determined by the reliability of data storage systems. However, modern storage systems are complicated. A storage system is typically comprised of many components, from hardware to software, and a problem can occur in any component. When it happens, a storage system may stop working and needs time to be repaired. A worse consequence could be the stored data is permanently lost due to unrecoverable errors. This would be a disaster for companies offering online services, such as Google or Amazon, because such a failure may cause large revenue loss or even trust loss from customers. Therefore, it is critically important to build reliable storage systems to ensure data reliability.

In a reliable storage system, data redundancy can be implemented on different layers. First, data redundancy can be implemented at hard disk layer. An example is Redundant Array of Independent Disks (RAID) [85]. A RAID system consists of a RAID controller and multiple hard disks. The overall RAID system is treated as a virtual single disk by operating systems. In the internal, the controller serves all data access requests, including writing data and reading data. User data is then stored across different disks. When one disk or more fails to work, the failure event can be detected by the RAID controller. On one hand, the RAID controller continues providing normal data services by using other surviving disks; on the other hand, the controller restores the data stored in the failed disks at background. RAID suffers several limitations. Firstly, RAID is a hardware solution, and thus it lacks of flexibility of configuring or upgrading the system easily during the system running. Secondly, all disks in a RAID have to be placed in a machine or a rack, which renders RAID unable to tolerate the failure of the machine or the rack. Thirdly, a typical RAID supports up to 36 disks, and it limits the scalability of a RAID.

Besides hard disk layer, data redundancy can be implemented at file system layer. In this implementation, a reliable file system is like a RAID controller, which stores data across multiple storage nodes. If the storage nodes are local hard disks, the file system is also referred to as software RAID. However, the storage nodes

can also be distributed on multiple sites. This can bring higher data reliability than RAID as it can tolerate a site failure. A common practice is that storage nodes are connected in a network. The network can be in a local area or in a wide area. During the system running, the client machines which are running user applications are connected to the network. When an application issues a data access request, the request will be received by the reliable file system. Then, the file system will communicate with storage servers to process it. After the process, the requested data will be returned to clients. The whole process is completely transparent to applications. Obviously, such an implementation makes the reliable system easily integrated into an existing system. Storage systems implemented in this way include HA-NFS [18], Zebra [52], Coda [97], Scotch [41], RAIF [65], HydraFS [106], PVFS [27], Panasas parallel file system [110], GlusterFS [43], and Lustre [58].

Besides file system layer, data redundancy can also be implemented at application layer. The difference from file system layer is that the data reliability services is provided by a library. Then, applications have to call a certain set of APIs to access the services. In the implementation of such a library, user data is also stored across multiple storage nodes for high data reliability. The major benefit of this layer is simplicity. This layer does not need to provide full file system services, but only a portion of them, and hence the library can be developed quickly and easily. Furthermore, a wrapper implementation can be based on the library and work as a file system, and the data services can be supported by the file system. This can ease the integration of the library. Although the file system may not be POSIX compatible, it may be sufficient for a wide range of applications. The examples of storage systems implemented in this way are RAIN [25], Harp [72], HYDRAstor [33], Google File System (GFS) [40], and Hadoop [103].

This chapter presents the design and implementation of a reliable file system, named *HyFS*, an essential component of a data storage system called *Hydra* [116]. *HyFS* is capable of employing general erasure codes and distributed storage servers to achieve high data reliability. When *HyFS* detects storage servers' failures, it will hide the failure from applications and continue its data service without any interruption. Then, at an appropriate time later, *HyFS* will automatically restore the data stored on the failed storage servers. As a result, data reliability is achieved. Furthermore, *HyFS* is implemented at file system layer so that the adoption of *HyFS* in a real system is easy and without any great effort. We believe that data reliability service is best to be implemented at file system layer and be full POSIX compatible. Regarding to data redundancy, *HyFS* supports a wide range of erasure codes, which can be simple replication scheme or complicated Reed-Solomon codes [96]. All these codes can be used simultaneously in a *HyFS* system. Hence, *HyFS* can be easily configured to meet different data reliability requirements. For performance, a

data request is served by multiple storage servers in parallel, which allows *HyFS* deliver high performance. Due to above reasons, we believe *HyFS* is an attractive solution for building reliable storage systems.

6.2 Related Work

Parallel Virtual File System (PVFS) [27] is a parallel system developed for high performance computing. PVFS is a cluster system containing both client and server side. PVFS can support multiple clients accessing the system simultaneously. At the server side, there are a manager server and multiple data servers. The manager server handles only metadata relevant requests, such as permission checking for open files. It does not participate in the normal data access operations. The data of a file is stored across multiple data servers. When a client issues a read/write request, it will directly communicate with those data servers. Then, the request will be served by the servers in parallel. This is a commonly used parallel access approach. PVFS is able to high performance and functionality for I/O-intensive applications, particularly for scientific applications. As the target of PVFS is high performance computing, PVFS does not implement any functions for data reliability.

Lustre [58] is another widely used high performance parallel file system. Lustre cluster has three kinds of systems: file system clients, object storage servers (OSTs), and metadata servers. The data of a file is stored on different OSTs, and then a data request is served in parallel by multiple OSTs. To achieve high performance, Lustre is implemented as a kernel space file system. This is different from PVFS, which is implemented as a user space file system. For high performance, Lustre even creates patches for ext3 file system. On one hand, these approaches greatly improve the performance of Lustre. On the other hand, they need nontrivial effort on installing Lustre. Lustre can support various protocols. For example, the storage servers can be over Fibre Channel (FC) or Serial Attached SCSI (SAS) connections. Lustre is a highly scalable file system. According to [78], Lustre can scale up to 25,000 clients. Regarding data reliability, Lustre does not provide rich functionalities. It depends on the underlying hardware or software RAID.

Panasas parallel file system [110] is a proprietary high performance file system. The file system contains storage nodes and manager nodes, and the ratio is about 10 storages nodes to 1 manager node. One distinction of the file system is the storage nodes implement an object store. An object contains both data and the data attributes. The storage nodes called Object Storage Devices (OSD) stores objects in a local OSDFS file system. The OSD is accessed through OSD specialized APIs. Besides high performance, the file system also provides high data reliability through a tiered parity architecture [64]. Horizontal Parity computes parity for

each object and writes the data and parity across multiple disks. If there is a failed disk, only the stored files need to be reconstructed from the remaining disks. Vertical parity is complementary to horizontal parity. It improves the internal ECC capabilities of the disk itself. Network parity can further detect errors in the data path between storage nodes and the client nodes. However, if a storage node fails, there could be data lost due to no protection on machine level.

Hadoop [103] is a popular cloud storage system. It is developed for the situations of massive amounts of data need to be processed. It contains several components. Two of them are Hadoop file system (HDFS) and Hadoop MapReduce. HDFS is a distributed file system that provides high throughput access to the stored data. Hadoop MapReduce is a software framework for distributed processing of large amount of data stored on HDFS. HDFS is a highly scalable distributed file system. It targets to store 10PB file data on 10,0000 storage nodes, and serve 100,000 clients. In the meantime, HDFS is also a reliable file system. For each piece of data, it stores three replicas of the data on three different storage nodes. In HDFS, the basic data unit is called block. Each file is divided into a number blocks with the same size. The typical block size is 64MB or 128MB. All metadata of files and blocks are managed by a server called master server. There are also several limitations on HDFS. HDFS is not suitable for a primary storage system due to high latency. HDFS needs to create three replicas for each block, resulting in high cost for hardware purchase and maintenance. Last, HDFS does not efficiently support several important functionalities, such as snapshot.

6.3 Design Goals

We have set multiple goals for *HyFS*. These goals guide through the design and implementation of *HyFS*. They are summarized below.

High Reliability To tolerate component failures, *HyFS* employs erasure codes to provide high data reliability. *HyFS* supports MDS erasure codes, as they use minimum storage overhead to provide the maximum reliability. Once *HyFS* detects a component failure, *HyFS* will hide the failure information and maintain the data availability as if the failure does not happen. During the failure recovery period, the data reliability level of *HyFS* decreases, but after the failure is recovered, the data reliability level is restored. Besides failure recovery, *HyFS* also uses erasure codes to perform error detection and correction, which further improves the system's reliability.

High Flexibility *HyFS* is designed to support any erasure code to be used. There are many erasure codes proposed and they mostly differ in three characteristics: reliability level, cost, and performance. In general, high redundancy means high reliability level, high cost, and low performance. On the other hand, low redundancy usually indicates low reliability level, low cost, and high performance. As a result, different erasure codes have different trade-offs, and there is no such a code that can fit into all environments. A developer of a reliable storage system should choose a code which best satisfies his requirements. To provide the flexibility of using any erasure code, *HyFS* offers a framework that allow any code to be easily plugged into the system.

File System Interface *HyFS* is a POSIX compatible file system. *HyFS* can be mounted on any Linux system and running as a native file system. This is different from other systems running at application layer. Running at file system layer brings significant advantages: 1) it facilitates developing new applications. As *HyFS* runs as a native file system, application developers can use their familiar file system calls to access files; 2) it makes legacy applications easy to run on *HyFS*. As legacy application have been developed for a long time, it becomes hard or even impossible to change them. But because *HyFS* is a native file system, any application running on existing Linux file systems can be seamlessly ported to run on *HyFS*; 3) it allows any applications to run on *HyFS*, even when its source code is not available. This is particularly useful for many valuable tools on Linux because their binary files can directly run on *HyFS*. Note that different from other highly available file systems, *HyFS* was targeted as a native file system when it was designed. Hence, many optimizations are implemented in *HyFS* so that file access requests can be served efficiently.

High Performance Performance is a critical consideration in *HyFS*. The traditional view of storage systems using erasure codes is that they cannot achieve comparable performance to others using replication scheme. Actually this point of view is arguable, because there is no any previous work quantitatively comparing the performance of these two different systems. Even though it is possible that there is performance gap between the two types of storage systems, their performance difference may not be significant. Therefore, we hope to address this issue by *HyFS*. As *HyFS* supports replication and other erasure codes, *HyFS* provides a platform to measure the performance of the two different systems and gives them a comparison.

Easy Deployment There are two aspects of easy deployment of *HyFS*. One aspect is *HyFS* is expected to be running on commodity hardware. This would allow building a reliable storage system with low cost. However, as commodity hardware is not as reliable as enterprise hardware, this imposes more challenges for

HyFS to provide high data reliability. The other aspect is that *HyFS* can leverage existing software to quickly build a reliable file system. For example, NFS is a popular network file system in a local area network. To protect the cost invested on NFS, *HyFS* can be configured to run on top of NFS, and furthermore, *HyFS* takes account of properties of NFS and contains implementations especially tailed for NFS.

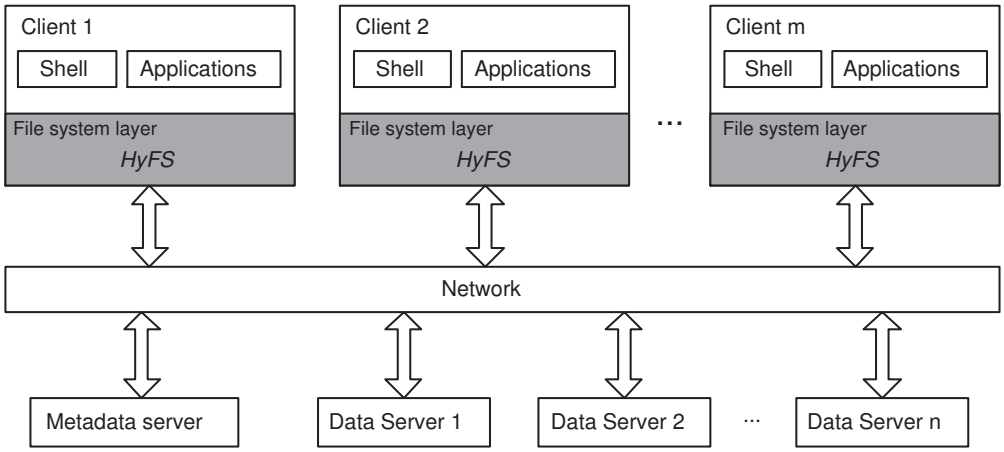


Figure 6.1: A Cluster of *HyFS*.

6.4 Overview of HyFS

6.4.1 Architecture

Figure 6.1 shows a cluster of *HyFS*. *HyFS* is a distributed storage system, which contains client side and server side. In the client side, *HyFS* is mounted as a file system, and applications on client machines can access *HyFS* just like other file systems. With this way, applications are completely transparent to the data reliability services provided by *HyFS*. In the server side, *HyFS* contains multiple storage servers, and they can be classified to two different types: metadata server and data servers. Regarding data communication, *HyFS* uses a simple protocol. That is, there is only data communication between clients and servers.

In *HyFS*, the metadata server contains the metadata information of the whole file system. There are two parts of metadata. The first part is the metadata of the hierarchical structure of the file system. The second part is the metadata of file, which includes the permissions of owner/group/others, the file size, the address of data servers, and etc. When an application initiates accessing a file and provides the file path, *HyFS* parses the path and locates the metadata of the file. Next, *HyFS* reads the file's metadata and obtains the file layout information. Hence, given a file path, *HyFS* contains the enough information to serve a file access request.

From application perspective, a user file is a logical file visible to applications. In *HyFS*, a user file is

comprised by multiple file fragments, and they are distributed on several data servers. Then, if an application issues a write request to a user file, *HyFS* will strip the file data to data servers and update the file's metadata accordingly. For a read request, *HyFS* will retrieve file fragments from data servers and assemble them together and then return the file data to the application. As a cluster of *HyFS* may contain hundreds of data servers, *HyFS* can aggregate the space of all data servers to provide a large capacity file system. Note that a file is not distributed to all data servers, but only on a handful of them. This depends on how the metadata server does the allocation.

As most part of *HyFS* is in the client side, we focus on this part in the rest text.

6.4.2 Components of *HyFS*

In the client side, *HyFS* consists of four major components, which are described in Figure 6.2. The components are discussed in details from Section 6.4.3 to Section 6.4.6.

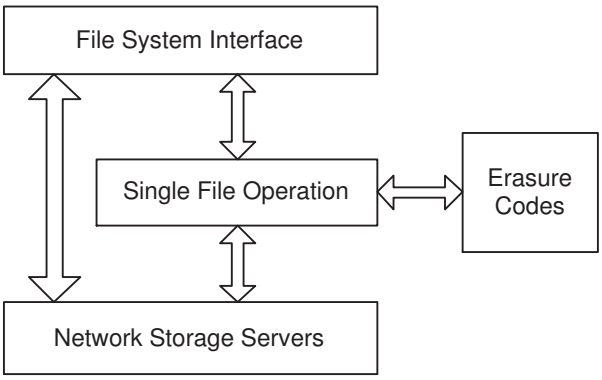


Figure 6.2: Components of HyFS.

6.4.3 Component: File System Interface

File System Implementation

FUSE As developing a stable file system at kernel space demands a great amount of time, we quickly implement a prototype of *HyFS* by using FUSE [104]. FUSE provides a framework to facilitate the development of a user space file system. It has been used in many well known file systems, such as SSHFS (based on SSH File Transfer Protocol) [1] and NTFS-3G (to access NTFS partition [2]). FUSE contains one module running in kernel space. This module deals with all complicated interactions with operating system at kernel space, so that the development of *HyFS* becomes relatively easy.

Figure 6.3 demonstrates how applications communicate with *HyFS*. Suppose an application issues a file

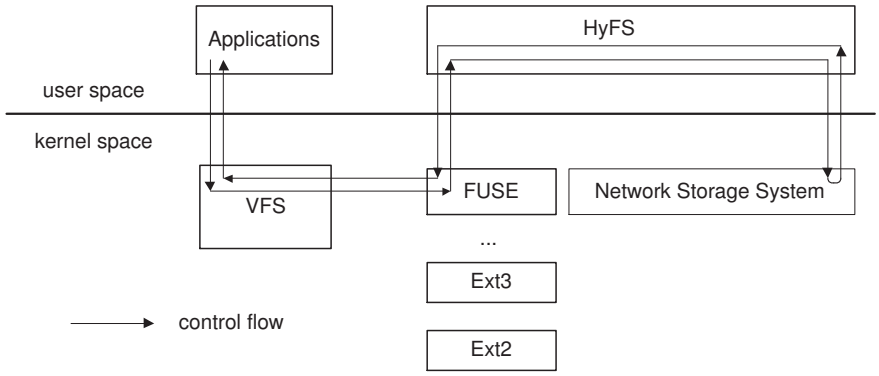


Figure 6.3: Communication between applications and *HyFS*.

access request through a system call (such as **read** or **write**). The system call would be caught by virtual file system (VFS), a core operating system component of Linux. Then, VFS finds which file system should process this request according to the file path; in our case, this is FUSE. As FUSE is working on behalf of *HyFS*, FUSE forwards the request to *HyFS*. Next, *HyFS* performs file operation for the request and returns the result to FUSE, and FUSE forwards the result to VFS, which passes it to the application. As such, The whole process for the request completes.

Performance Overhead The use of FUSE in *HyFS* may cause nontrivial performance overhead. One overhead is from mode switch between user space and kernel space. Serving one file request involves totally six mode switches. As mode switch is a time consuming operation, it may cost a few thousands of CPU cycles. Another overhead is from data copy. To respond to either a write request or a read request, there need four data copies: 1) one is between applications and VFS, 2) one is between VFS and FUSE, 3) one is between FUSE and *HyFS*, 4) the last one is between *HyFS* and network storage servers. The overhead of data copy may be significant, particularly when large files are accessed. Therefore, it is important to quantify the two overheads and measure their impacts. We present our experiment results in Section 6.6. A simple conclusion is that we may have to implement *HyFS* at kernel space for better performance.

Important Issues

Security *HyFS* is a secure file system. It can protect the stored files from unauthorized access. Each directory or file in *HyFS* has ownership and permission control. *HyFS* implements it by taking advantage of the file system in the metadata server. For any file, *HyFS* stores its metadata as a regular file in the metadata server, and *HyFS* relies on the local file system of the metadata server to grant a permission or deny a file request.

Data Corruption A robust file system should maintain file system consistency even if the system crashes during performing writing operations. A common approach is using journal. *HyFS* implements this function by relying on the journal support of metadata and data servers, which can run local file systems with journal functionality. Hence, journal would be provided by *HyFS* for free. Furthermore, as *HyFS* uses erasure codes for data reliability, *HyFS* can leverage the inherent error detection and correction of erasure codes to perform data corruption test and recovery. This would bring even higher data reliability.

6.4.4 Component: Single File Operation

In *HyFS*, a basic question is how to support operating a single file. This question is addressed by two components: Single File Operation and Erasure Codes. We discuss the former component in this section.

Overview

This component supports all regular file operations, including opening/reading/writing/seeking/closing a file. We define a set of APIs in Figure 6.4. Obviously, they are very similar to their counterparts in C library.

```
// correspond to open function
HFILE  *hopen(
      const  char  *pathname,  int  flags, mode_t mode);

// correspond to read function
ssize_t hread(
      HFILE *h_file,
      void  *buf, size_t count);

// correspond to write function
ssize_t hwrite(
      HFILE *h_file,
      const  void  *buf, size_t count);

// correspond to lseek function
off_t hlseek(
      HFILE *h_file,
      off_t offset,   int  whence);

// correspond to close function
int  hclose(
      HFILE *h_file);
```

Figure 6.4: APIs of file operations.

HFILE is a file handle of *HyFS*. It is like a file descriptor. The file handle contains all necessary information to access a file, such as erasure code, and data server locations etc. **HFILE** has to be initialized by **hopen** and then passed to all other APIs. **hread** collects file fragments from multiple data servers and combines them together. Then, it returns the data requested by applications. If a failure occurs to one data

server or more, **hread** will perform decoding operation and hide the failure. Similarly, **hwrite** performs write operations, and **hclose** closes the file handle and frees all allocated resources.

Important Issues

Data Buffer For most high performance file systems, data buffer is necessary to bridge performance gap between main memory and persistent storage. Currently, *HyFS* does not implement data buffer in itself; instead, it relies on the file system exposed by network storage servers, such as NFS. That means, **hwrite** will write data to those file systems immediately when it accepts data and **hread** will read out only the minimal data that is requested. The assumptions is that those file systems, such as NFS, should efficiently manage data buffer just like they are supposed to do. Hence, any additional data buffer in *HyFS* seems unnecessary. In a rare case, if the performance of such a file system is poor, we may add a buffer layer for that file system, but may not for general ones.

Concurrent Access In practice, when a file is opened by multiple applications, they may issue file access requests at the same time. This situation is concurrent access. As the requests could be interleaved, they may result in inconsistency to the file. *HyFS* deals with this issue by following the convention of Linux file system. That is, *HyFS* leaves the work of maintaining consistency to the application developers. This choice is based on two reasons. First, we have tested ext3 file system, and we found that if multiple applications write a single file simultaneously, there could be inconsistency in the file. Hence, we believe that this is a convention of Linux file system, and this behavior should be familiar to experienced application developers. Second, *HyFS* can not fully understand the intention of applications but their developers do. Therefore, the best place to do the concurrent access control is at application layer.

6.4.5 Component: Erasure Codes

The Erasure Codes component implements all functions relating to the use of erasure codes. This component provides two basic services, encoding and decoding. When *HyFS* responds to a write request, the data to be written goes through an encoding procedure; conversely, after data is read out for a read request, decoding will be performed. Note that if there is no any erasure, decoding will be skipped because user data can be directly read out for systematic erasure codes. To support various erasure codes within *HyFS*, this component defines a set of interfaces, shown in Figure 6.5. The interfaces have to be implemented for an erasure code before it is used in *HyFS*.

```
// perform init operation
void  code_init(
        code_data_t *data,
        int  n,  int  block_size);

// perform encoding operation
int  code_encode(
        code_data_t *data,
        char  *msg,  char  *parity);

// perform decoding operation
int  code_decode(
        code_data_t *data,
        char  *msg,  char  *codeword,   int  *lost);

// perform close operation
void  code_close(
        code_data_t *data);
```

Figure 6.5: Interfaces implemented by erasure codes.

Overview

Currently, we have implemented several erasure codes in *HyFS*. This is to demonstrate that it is easy to add an erasure code to *HyFS*. The implemented codes include replication scheme, the STAR code [54, 76], and Reed-Solomon Codes. Thus, the defined interfaces provide an open framework to support any erasure code in *HyFS*. We discuss each interface briefly as follows:

code_init This function contains three parameters. The first one is a pointer, where an erasure code can perform any necessary initialization and fill it with any content. The other two parameters are the number of data servers and the block size of a symbol in each codeword.

code_encode It performs encoding operation. The input parameter *msg* specifies the input data for encoding. Parameter *parity* points to the encoded parity data. The *msg* data and the *parity* data form a complete codeword.

code_decode This function performs decoding operation, the inverse operation of *code_encode*. This function is invoked when a failure is detected. Among the parameters, *codeword* is the pointer of a codeword, and *msg* is the pointer of decoded data. *erasure* indicates which symbols of *codeword* are erasures.

code_close When an erasure code is not used any more, this function should be called to free all allocated resources.

Efficient Encoding and Decoding

In *HyFS*, one performance overhead of erasure codes comes from encoding and decoding. Encoding is performed constantly when new data is written to the system, and decoding is only invoked when data is read out and erasures exist. Because these two operations bring extra overhead to *HyFS*, their performances have to be optimized to avoid being performance bottleneck.

Compared to decoding, encoding is a more frequent operation as it takes place whenever new data is written. We have proposed an efficient encoding algorithm named *DWG* algorithm for XOR-based erasure codes [77]. This algorithm schedules the XOR operation for erasure codes, so that the encoding can be performed more efficiently than that of the traditional algorithm. We are going to apply *DWG* algorithm to *HyFS* and see how this algorithm helps *HyFS* to achieve high performance.

Although decoding is not as frequent as encoding, its performance is still critical for *HyFS*. Decoding is invoked when any failure happens. This is the time of *HyFS* in performance and reliability degraded period. If decoding is slow, the system throughput is low, and more time has to be taken to restore data reliability. Hence, decoding efficiency affects both system performance and data reliability. For RAID-6 codes, we have proposed an algorithm, named *SCAN* algorithm for efficient decoding. We plan to apply this algorithm to *HyFS* as well.

6.4.6 Component: Network Storage Servers

At the bottom of Figure 6.2 is the component of Network Storage Servers. In fact, *HyFS* supports storing data on different types of storage nodes. A storage node could be a hard disk, a flash drive, or a storage server. Here, we simply use storage servers in a network to illustrate this component.

Overview

This component is delegated to access network storage servers for writing and reading data. It can be based on the locally mounted file systems exposed by network storage servers. This allows use standard file system calls to communicate with network storage servers. Then, *HyFS* can access metadata servers and data servers as if it accesses local file system, so that the implementation of this component is easy. For example, if NFS is exposed at each data server, we will setup mount it client on client machines, and this component will be running on top of NFS. However, if there is no such a facility provided by storage servers, we may have to develop a file system interface to wrap the data services of storage servers.

Variety of Network Storage Servers

HyFS can run on top of various types of network storage servers. On one hand, *HyFS* can utilize existing network storage servers in local area network, such as **NFS** and **AFS** servers. In this respect, *HyFS* is a stackable file system [118] as it is built upon other file systems. The benefit is that *HyFS* can leverage existing mature network file systems. On the other hand, *HyFS* can be based on storage servers in wide area network, such as storage servers accessible through HTTP, FTP, or GridFTP. This allows *HyFS* to run on Internet and access storage servers geographically distributed. It greatly improves data reliability, because even a catastrophe takes place at one site, user data can still be reconstructed from servers at other surviving sites.

Important Issues

Because storage servers in local area network and those in wide area network have different characteristics of latency and throughput, these differences may result in different performance concerns in this component. An example is when using remote Web servers as storage servers, the throughput cannot be comparable to that of servers in local area network. In this case, employing a large buffer is likely to be necessary to hide the high latency and low throughput of the Web servers. Our purpose here is to provide high performance for *HyFS* even when slow storage servers are used.

6.5 Critical Designs

In *HyFS*, there are several critical designs, which are discussed as follows.

6.5.1 Hierarchical Structure

For applications, *HyFS* presents a hierarchical structure of directories and files. In the internal of *HyFS*, we have several options to implement it.

Metadata Server

We term the hierarchical structure visible to users as *logical structure*. *HyFS* represents the *logical structure* by the structure of metadata files on metadata server. The metadata files are stored in local file system of the metadata server, and *HyFS* uses that file system to manage the logical structure. Hence, for each directory or file in logical structure, there is a corresponding directory or file in the metadata files

structure. For instance, to server a request of creating a directory in logical structure, a real directory will be created in the metadata server.

Data Servers

Different from metadata files, data files stored on data servers may not have the same hierarchical structure as logical structure. Instead, as metadata server contains all metadata of the file system, data servers have more flexibility on the structure of data files. Here, we present three mapping options from logical structure to the structure of data files. The three options differ in the costs of mapping a file and updating the structure.

Identical mapping On all data severs, the structure of data files is completely identical to logical structure. Then, when a user file is accessed, the location of its data files can be easily found as their structures are identical, and hence this mapping approach has minimum mapping cost. However, it may cause high cost of updating the structure. For instance, when a user directory is renamed, the directory on data servers also needs to be renamed. Furthermore, the file paths of the data files under that renamed directory are changed, and thus all relevant metadata files have to be updated so that they can point to the new data file paths. Such an update may result in a significant performance overhead when the directory to be renamed contains many subdirectories and files. Nonetheless, due to its simplicity, this approach is used in the current implementation of *HyFS*.

Random mapping Compared to identical mapping, this approach goes to another extreme. In this approach, the structure of data files has no fixed relationship with logical structure. For example, creating a directory in logical structure does not indicate creating a directory in the structure of data files. Nonetheless, one logical file still corresponds to one data file in one data server. This approach intends to minimize the cost of structure update. For example, renaming a logical directory could only need an update to the metadata server and no other operation is required on data servers. However, this approach needs to have a table to maintain the mapping between the two structures, which increase the mapping cost.

Isomorphic mapping This approach stays in the middle of above two approaches. It keeps the structure on data servers isomorphic to logical structure. That is, there still exists one-to-one correspondence between the structures for directories and files, but they are not required to have exactly same names. This approach could balance the structure mapping cost and structure update cost. For example, To serve a directory renaming

request, *HyFS* only needs to update the mapping rule for that directory.

6.5.2 Efficient Read and Write

For any single file, *HyFS* presents its content to applications as a byte stream, but in the internal implementation, a file is treated as a codeword stream. Such a difference makes it difficult to efficiently support read, write, and seek operations. This section describes how *HyFS* addresses this issue.

Overview

Due to the use of erasure codes, a codeword is the smallest data unit in *HyFS*, and a file is considered as a codeword stream. The codeword size depends on several parameters, including erasure code, n, k, and ps (packet size). However, for applications, *HyFS* presents a file as a byte stream like a conventional file system. This raises a couple of challenges. First, *HyFS* has to support byte stream operations based on codeword stream. Secondly, *HyFS* cannot assume any file access pattern from applications, and read/write/seek requests could randomly occur in a file request sequence. Lastly, *HyFS* should support file operations efficiently for performance reason. All these factors complicate the implementation of *HyFS*.

In *HyFS*, we classify file access patterns into two categories: sequential access and random access. We first introduce the two patterns, and later we give a simple heuristic to determine which one is the most likely pattern for a workload. Note that the pattern is detected for one file handle, not for one file.

Sequential Access Pattern

When an application accesses a file, the application may issue a sequence of file access requests. If the requests show a strong spatial locality in terms of their accessing positions in the file, this sequence is considered to have a sequential access pattern. For example, when we issue a *cp* command, *cp* accesses the file content of both files sequentially to copy data, and hence the file requests have sequential access pattern. We propose using an automata to handle this pattern. The automata is shown in Figure 6.6.

In the automata, there are three states: write, read, and seek. At one time, *HyFS* is in one state. The initial state is seek, and the state change is decided by the type of file request. For example, suppose the current state is write. If the next file request is read, then the state would be changed to read. For write and read state, we keep a write and read buffer. The buffer size is exactly one codeword size. The write or read buffer stores the most recently accessed codeword. As sequential pattern demonstrates strong spacial locality, all content of the codeword in the buffer is likely to be used. If we don't have such a buffer, then even when we

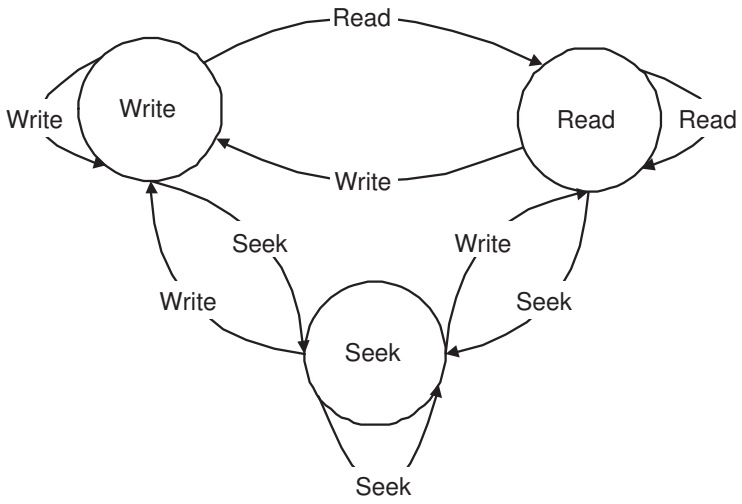


Figure 6.6: An automata to handle sequential access pattern.

write/read a byte, we have to perform a write/read operation. This would be inefficient.

The state of seek is important in the automata. This is used to invalidate buffer content. As write/read and seek requests can be randomly issued by applications, *HyFS* does not have any assumption of the request sequence. For example, if an application reads file content in one file offset, and then it moves to another file offset to perform read. Consequently, the read buffer for the first read cannot be used for the second read. Then, *HyFS* uses state seek to invalidate the read buffer when it finds that file offset is changed across file requests. For performance reason, not every seek request has to cause state change. For example, if a seek request only moves the file offset within a codeword, then the state is not changed.

Random Access Pattern

Different from sequential access pattern, random access pattern does not show any spacial locality. For two consecutive read or write requests, the file offsets could be far away from each other. Then, if *HyFS* writes or reads a complete codeword and uses a buffer to hold it, the performance may be low because only part of a codeword is used. To address this type of pattern, we use a different approach to handle it. The idea is each request is processed stateless. There is no buffer or state to be maintained across different requests. Rather, for one request, we only perform minimal and necessary work. For example, if a file request intends to read 5 bytes, *HyFS* will read exactly 5 bytes regardless of the codeword size. The similar operation is performed for a write request. Although this approach looks simple, we find it is highly effective for random access pattern.

Compared to the approach for sequential access pattern, the approach for random access pattern uses a

different set of APIs provided by the component of Erasure Codes. This is because the approach for random access pattern works on writing or reading partial codeword. For a read request, *HyFS* simply goes to read the data that are requested. However, for a write request, the process becomes complicated. When a new piece of data is written to a user file, the parity data has to be updated. As this approach performs minimal write on user data, it also tries to minimize the update to the parity data. Hence, *HyFS* needs to calculate carefully that which part of parity data should be modified. Such a calculation depends on the component of Erasure Codes.

Pattern Determination

For a file handle, *HyFS* constantly keeps track of the file requests issued on it and then determines the most likely access pattern. Currently, *HyFS* uses a simple method to do the determination. This method maintains a window that saves most recently 100 file requests. If within the window, *HyFS* finds that there is less than 20% of seek requests that move file offset across different codewords, the requests are considered to have the sequential access pattern; otherwise, they follow random access pattern. This is a simple heuristic, and we believe that there is still great potential to improve it.

6.5.3 Load Balancing

Load balancing is about how to evenly distribute file access requests to data servers, so that all of them receive similar amount of work. This is to improve system throughput. Here, we focus on load balancing for a single file, not system wide though. We use a technique called rotation [85]. Rotation means that different codewords may have different layout on data servers. The purpose of rotation is to make a data server contain both data and parity data of a file so that each server has the same probability to serve an arbitrary file request. Hence, file access requests are likely to be evenly distributed to data servers.

In *HyFS*, we implement rotation in component File System Operation. This choice has two advantages. One is such a rotation is completely transparent to component Erasure Codes, and thus it only needs to be implemented once but can be used for all erasure codes. The other one is the flexibility of rotation. Although round-robin is a commonly used rotation approach, other approaches may be preferred for a specific environment. As rotation is implemented in one place, component File System Operation, adopting a different rotation approach is easy.

6.5.4 Failure Detection

As *HyFS* runs on commodity hardware, storage servers may fail at anytime, and then it is important for *HyFS* to detect a failure event. By breaking down this issue, we find there are three relevant problems: 1) how to detect a failure, 2) when to perform failure detection, and 3) what to do with failed storage servers. We discuss them in details.

In *HyFS*, storage severs may expose various types of file systems, which makes it hard to find a general way to detect failures efficiently. There could be two different approaches to do it. One approach is to use $open$ file system call to open files in the storage servers. If the return value is 0, then the storage server is alive; otherwise, the storage server is considered as a failure. This is a simple method, but it has drawbacks. For instance, if an NFS server is hard mounted (this is a usual way) and the server fails accidently, $open$ will be blocked and the whole file system will hang there. Another approach can solve this problem. That approach explicitly implements failure detection rather than depends on $open$. It first gets file system's types provided by each storage server. Then when it needs to detect an NFS server, it initiates a socket connection with the server. If the connection can be established, then there is no failure with the server. Otherwise, the server is treated as a failure.

Another problem is when to perform failure detection. One choice is doing it in a fixed interval. A thread can be generated when *HyFS* is stared. Then, the thread connects all storage servers regularly and finds those failed ones. However, one shortcoming of this approach is that if a storage server fails during two consecutive intervals, the failure will not be detected when it happens and any issued file access requests may be blocked. To solve this problem, another approach performs failure detection whenever there is access to storage servers. If a failure happens, the failure will be detected during the access and propagated to *HyFS*. One main concern of this approach is performance. As every access has to perform failure detection, the cost may be high, and thus this approach should be implemented efficiently. Therefore, a complete solution should combine the above two approaches, so that it can detect failure timely and efficiently.

The last problem is what to do with failed storage servers. If a storage server is detected as a failure, there could be different causes behind it. If it is because this server meets a permanent failure, then the data stored on the storage server should be recovered. This may result in long failure repair time. However, if this failure is caused by power outrage or network card malfunction, the failure could be fixed quickly by replacing power outlet or network card. In this case, the failure repair time is relatively short. One issue is when the storage server comes back, the data in the server may be stale and cannot be used anymore. In

current implementation of *HyFS*, if a server failure is detected, it always considers the server as a permanent failure, and the failed server will be recovered in background.

6.6 Performance Evaluation

This section presents performance results of *HyFS*.

6.6.1 Overhead of FUSE

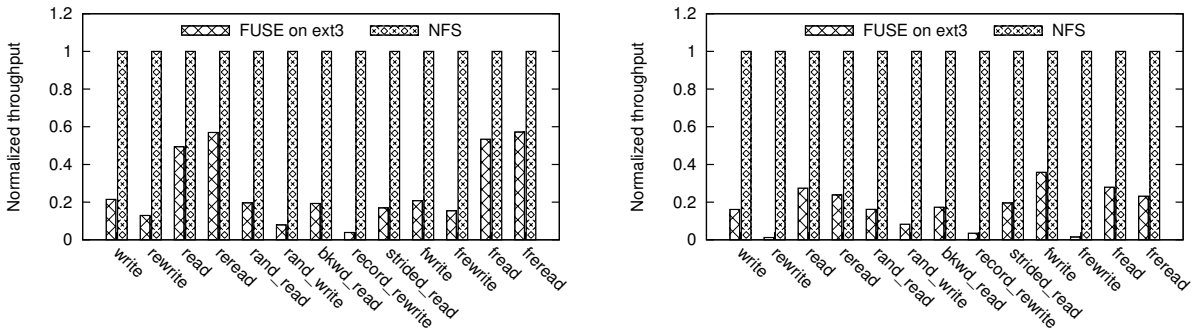
As mentioned in Section 6.4.3, FUSE allows quick development of a file system, but it also incurs extra performance overhead. We conducted several experiments to quantify the overhead. Here, we use a file system, called *fusexmp*. fusexmp is a file system provided by FUSE installation package [104]. fusexmp is a stackable file system. It has to run on top of another local file system, which is referred to as the host file system of fusexmp. Then, for any file access request, fusexmp simply forwards it to the host file system without any process.

The overhead of FUSE is measured as follows. We run two different types of tests. The first type of test runs a benchmark software on a host file system directly. Another type of test runs on fusexmp. Hence, by comparing the performance results from the two type of tests, we are able to measure the performance overhead caused by FUSE. It is worth noting that the performance overhead is affected by many factors, such as the host file system, the workload, etc. Because of it, we run several tests to get a comprehensive view of the overhead.

We employ a widely used benchmark software Iozone [3]. Iozone can measure the throughput of a variety of single file operations, including write, rewrite, read, reread, random read, random write, backward read, record rewrite, stride read, fwrite, frewrite, fread, and freread.

Ext3

This experiment uses ext3 as the host file system for fusexmp. To simulate different workloads, the experiment uses two size pairs: [20MB, 100KB], and [200MB, 100KB]. In the bracket, the first number is the file size, and the second one is the record size. Because we focus on the performance overhead of FUSE, we don't present the absolute throughput values; rather, we give the comparison of normalized throughput. The normalized throughput of ext3 is set as 1, and the normalized throughput of fusexmp is calculated accordingly. The performance results are shown in Figure 6.7.



(a) ext3 with fs=20M, rs=100K

(b) ext3 with fs=200M, rs=100K

Figure 6.7: FUSE overhead measured on ext3

We can make the following observations from Figure 6.7.

1. For all operations, the performance of FUSE on ext3 is much worse than that of native ext3. This indicates the performance overhead of FUSE is not trivial, but significant.

2. The overhead is different across operations. Commonly, the overhead on write operations is greater than that on *read* operations. For example, the performance overhead on *write* is two times of that on *read*.

3. The performance of most operations under workload [200M, 100K] is lower than that under workload [20M, 100K]. It means the overhead is sensitive to the workload. In fact, workload [200M, 100K] invokes more disk I/Os than workload [20M, 100K], and then the performance overhead of FUSE on this workload should be less obvious considering the overhead of FUSE only relates to CPU cycles and data copy. But we observed the different thing. We'll take more time to investigate this issue.

NFS

The next experiment uses NFS as the host file system for fusexmp. As NFS is a network file system, all file requests are actually served across network. Hence, this measures how network affects the performance overhead of FUSE. In the experiment, there is one NFS client and one dedicated NFS server. Both are running NFS V3. The two machines are connected in a 100Mb/s network. The performance results are displayed in Figure 6.8.

Figure 6.8 shows two trends. First, the performance of fusexmp on NFS is worse than that on ext3. It implies the performance overhead is also sensitive to the host file system besides workload. Second, all write performances of fusexmp on NFS are extremely low. The reason could be that FUSE may cause

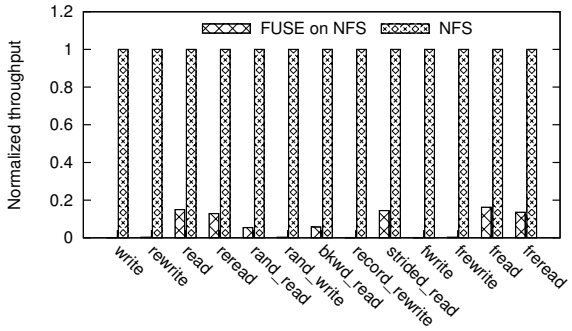


Figure 6.8: FUSE overhead measured on NFS

performance overhead other than that from mode switches and data copy. This observation is related to the third observation we found for fusexmp on ext3. We will investigate the two issues together in future.

6.6.2 Micro Benchmark

Now we present the micro benchmark of *HyFS* using Iozone. In the following experiments, the cluster of *HyFS* contains five machines, including one client machine and four delegated data servers. The metadata server is running at the client machine. The network bandwidth is 100Mb/s.

Read and Write

The first experiment measures the performance of read and write. As mentioned in Section 6.5.2, file requests could have sequential access pattern or random access pattern. To efficiently handle the two patterns, this chapter proposes three algorithms. The first one (algorithm *seq*) is designed to process sequential pattern, and the second one (algorithm *rand*) processes random pattern. The last one (algorithm *auto*) is a hybrid, which dynamically determines the pattern and then applies the proper algorithm to handle it. It uses a simple pattern determination method mentioned in Section 6.5.2. We present their performances in Figure 6.9.

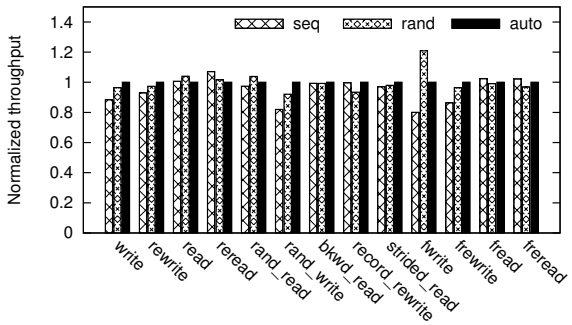


Figure 6.9: Performance of different algorithms

There are several observations made from Figure 6.9. First, for most write operations, algorithm *auto* is the fastest one followed by algorithm *rand*, and algorithm *seq* is the slowest one. Second, for most read operations, algorithm *auto* is still the best one, and the next one is algorithm *seq* followed by algorithm *rand*. As a result, algorithm *auto* is the most efficient one among all three algorithms. It is worth noting that algorithm *rand* outperforms algorithm *auto* for *fwrite* operation. This implies that the pattern determination approach of algorithm *auto* can be improved further.

Scalability

The second experiment considers the scalability of *HyFS*. Here, scalability means how the system performance scales as the number of data servers increase. We did two tests. Both of them are using single parity code. One test is *(2, 1)*, which uses two data servers. Another one is *(4, 3)*, which uses four data servers. The performance result is shown in Figure 6.10.

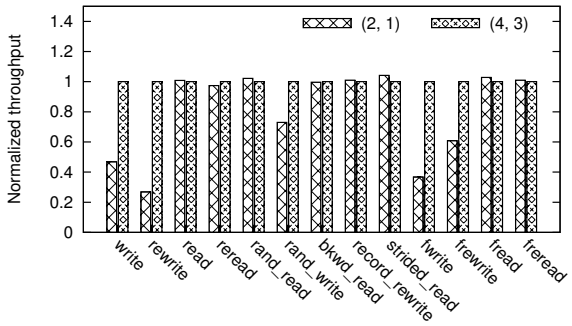


Figure 6.10: Performance of HyFS with (2,1) or (4,1)

Figure 6.10 shows that the performance of *(4, 3)* is much better than *(2, 1)* for all write operations. This is expected as *(4, 3)* has double number of data servers. However, the read performance of both configurations is similar. The reason could be Iozone writes a file first, and then uses the file for read test. Then, the content read by Iozone is mostly from cache, and hence it does not scale with the number of data servers.

Erasure Codes

The next experiment measures how various erasure codes impose different performance overhead on *HyFS*. We have three configurations in the experiment. *rs (4, 3)* uses Reed-Solomon code (4, 3), and *parity (4, 3)* uses parity code. Both can tolerate one data server failure. *dist (4, 4)* uses distribution code, without any fault tolerance. The performance is shown in Figure 6.11.

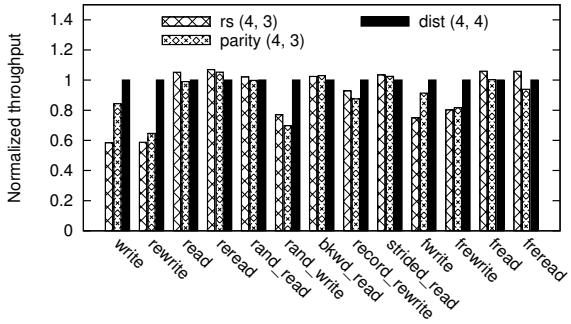


Figure 6.11: Performance of HyFS with various erasure codes

Figure 6.11 shows for all write operations, *dist (4, 4)* achieves the best performance followed by *parity (4, 3)* and *rs (4, 3)*. This is expected. The worse performance of *parity (4, 3)* and *rs (4, 3)* attributes to the additional parity data written by them. When it comes to the performance gap between *parity (4, 3)* and *rs (4, 3)*, although they have the same amount of data to write, the computation of parity code is more efficient than Reed-Solomon codes, and hence *parity (4, 3)* outperforms *rs (4, 3)*. Another observation is for all read operations, the performances of the three configurations are similar. This is because in this case, the performance is determined by the amount of data to read. As there is no failure, erasure nodes do not bring any performance overhead.

6.6.3 Real Applications

To have a more comprehensive view of *HyFS*, we run several real applications on *HyFS*. There are two purposes of doing it. On one hand, it shows that *HyFS* runs as a file system, which can support existing applications without any effort. On the other hand, it measures the performance of *HyFS* from application perspective.

Apache HTTP Server

In this experiment, there is a machine running Apache HTTP server. The root directory of the virtual host contains three directories, and they are on three different file systems: ext3, NFS, and *HyFS*. Each directory contains a file with the same content. The file size is 2GB. Then, we use two benchmark tools – ab [4] and httperf [5] – to measure the time of retrieving a file from the Apache server to another machine. For each tool, we access the files in the three different directories and compared the time of accessing the file. Hence, this workload represents downloading a large file from a Apache HTTP server.

Figure 6.12 shows that the transfer rates of all file systems are close for both benchmark tools. Actually,

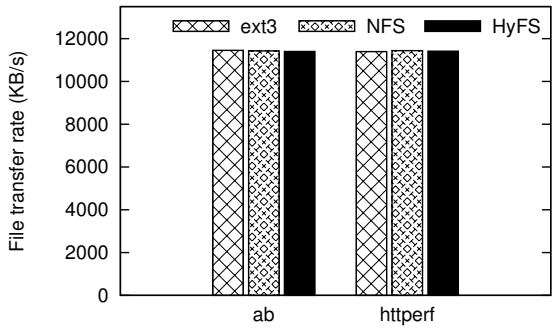


Figure 6.12: Performance of Apache on various file systems

the performance bottleneck is in the network. As the network bandwidth is 100Mb/s, the transfer rate of 12000KB/s indicates the network is almost saturated. Therefore, this experiment shows that *HyFS* provides high performance for Apache HTTP server when it accesses large files. Regarding small files, we found that they are cached in memory after they are accessed once, and hence the performance of file systems is relatively unimportant in this case.

MySQL Server

We did another experiment using MySQL. MySQL stores user data, and hence data reliability of MySQL is critically important. To achieve high reliability, the database files of MySQL can be stored in *HyFS*. Due to the fault tolerance capability of *HyFS*, the data reliability of MySQL is ensured. Like what we did in the above experiment, we create three databases, and they are on ext3, NFS, and *HyFS* respectively. Then, we use two benchmark tools –SysBench [6] – to measure the performance of MySQL when it is running on different databases.

SysBench SysBench [6] provides an impression of OLTP performance of a database server. Figure 6.13 displays the measured results in our experiment. The figure shows that the performance of OLTP on three file systems are close. This indicates that *HyFS* can provide high performance for OLTP of MySQL.

6.7 Summary

This chapter proposes a file system called *HyFS*. *HyFS* employs erasure codes and network storage severs to build a reliable file system. As *HyFS* distributes data to storage severs with redundancy, *HyFS* can tolerate a certain number of servers failures. When a failure event happens, *HyFS* hides this information and

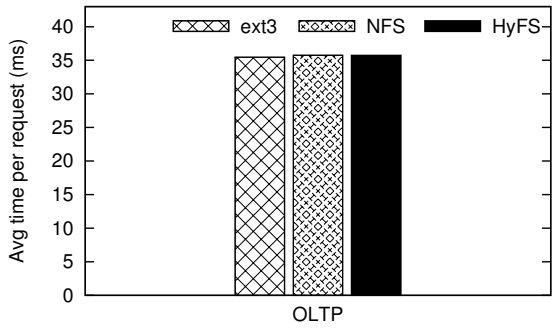


Figure 6.13: Performance of SysBench on various file systems

continues providing data service. In a proper time later, *HyFS* will recover the failed data and restores the system's data reliability.

In this chapter, we first described the architecture of *HyFS*. *HyFS* is a cluster system, which contains client side and server side. In the client side, *HyFS* is running as a regular file system, which greatly facilitates the adoption of *HyFS* in a real system. Then, we briefly introduced the components in *HyFS*. We provided a framework to support various erasure codes to be used in *HyFS*. Next, we presented several critical designs in *HyFS*. Particularly, we introduced two different file access patterns and addressed how to efficiently support file operations for the two patterns. Lastly, we presented the results of performance evaluation.

CHAPTER 7 Conclusions and Future Directions

7.1 Conclusions

This dissertation studies how to implement a reliable file system by using erasures codes. It focuses on five relevant problems.

This dissertation first proposes several algorithms to perform encoding operations efficiently for XOR-based erasure codes. We observed that the order of XOR operations is flexible and can significantly impact encoding performance . Using this observation, we proposed three new XOR-scheduling algorithms that have different characteristics in the XOR scheduling. We have shown that the new algorithms greatly outperform the traditional algorithm, and they are applicable to multiple erasure codes.

Then, an efficient decoding algorithm for RAID-6 erasure codes is presented. A RAID-6 system can tolerate up two entire disk failures. In such a system, each disk may encounter entire disk failure or sector failures. Consequently, disk failures in a RAID-6 system could become complicated. We present a general decoding algorithm named *SCAN*, which can recover all disk failure cases efficiently. Different from another well known algorithm called *Matrix Method*, *SCAN* uses Tanner graph on decoding. We conducted experiments on various RAID-6 codes, and shown that *SCAN* achieves much better decoding performance than *Matrix Method* in all failure cases.

Next, this dissertation provides an efferent error correction algorithm for the STAR code. The motivation is to combat disk errors by using error correcting codes. This is because redundancy is already in place introduced by error correcting codes to cope with failures at system level, and then we advocate using error correcting codes and overcome both failures and silent errors simultaneously. We propose an efficient error decoding algorithm named EEL (Efficient Error Locating) for the STAR code, which can tolerate up to three disk failures. The EEL algorithm leverages the geometric structure of the STAR code to obtain high performance. The performance results show that the decoding performance of EEL algorithm is significantly better than that of a naive algorithm.

Besides dealing with data reliability, this dissertation introduces an efficient implementation of arithmetic operations for large finite fields. With the advent of cloud storage, a whole host of new failure models need to be considered, e.g., mis-configuration, insider threats, software bugs, and even natural calamities.

Accordingly, storage systems have to be redesigned with robustness against adversarial failures. Finite fields are widely used in constructing error-correcting codes and cryptographic algorithms. To build secure storage systems, we have most interest on the finite fields from $GF(2^{32})$ to $GF(2^{128})$. We are concerned with two arithmetic operations: multiplication and division. By using extension field method, we develop efficient implementations for large finite fields.

Lastly, this dissertation presents the design and implementation of a reliable file system, named *HyFS*. *HyFS* employs general erasure codes and distributed storage servers to achieve high data reliability. When a failure happens to storage servers, *HyFS* will detect the failure timely and hide the failure from applications. The data service will be continued without any interruption. Then, at an appropriate time later, *HyFS* will restore the data stored on the failed storage servers. Furthermore, *HyFS* is implemented as a native file system so that the adoption of *HyFS* in a real system is easy. Regarding to data redundancy, *HyFS* supports a wide range of erasure codes running simultaneously, and hence *HyFS* can be easily configured to satisfy data reliability requirements of different environments. In terms of performance, *HyFS* serves a data request by multiple storage servers in parallel, leading to high relievable performance.

7.2 Future Directions

We found that there is still a lot of future work to do with *HyFS*. Here we only list three important things: the design of a new set of erasure codes, the development of new file systems for storage servers, and the scalability to support a cloud storage system.

7.2.1 Erasure Codes

When an application issues write requests to a file, *HyFS* has different behaviors for sequence access pattern and random access pattern. If write requests show sequential pattern, *HyFS* will read out the affected codewords and perform update operation. As *HyFS* has complete codewords in main memory, the encoding operation is easy. Furthermore, as the updated codewords are in main memory, the update to the file is also simple. However, for random access, both operations of encoding and file update become complicated, because it can only touch the part of codewords that has to be updated. In this case, as *HyFS* aims to perform minimum update for the write requests, then *HyFS* needs to calculate precisely which part of symbols should be touched. This involves non trivial implementation complexity for most existing erasure codes. In addition, updating parity symbols in a minimal way is also hard to implement. For example, if the code used

in *HyFS* is EVENODD, an update to consecutive data symbols within a codeword may result in multiple scattered updates to parity symbols. If the code is Reed-Solomon codes, then the update may also not be minimal due to the computation of Reed-Solomon codes on finite fields different from $GF(2)$. Therefore, we may need new erasure codes that can efficiently support write requests with random access pattern and the implementation should also be easy.

7.2.2 File Systems on Storage Servers

In a network environment, *HyFS* uses NFS to build a cluster system. As NFS is a popular network file system, this is a quick way to develop *HyFS*. However, we found that this way has several limitations. Firstly, NFS servers are recommended to be hard mounted for critical data because this way makes sure that each write operation succeeds after write returns. However, if a hard mounted NFS server meets a failure, the NFS server will hang and cause *HyFS* to freeze. To solve this issue, *HyFS* pings an NFS server regularly to detect the server is accessible. Although this approach works for NFS servers, it is not a general way to perform failure detection. Secondly, if an NFS server meets a temporary failure and comes back later, *HyFS* has no way to be automatically notified. This is obviously an inefficient approach. Thirdly, as *HyFS* uses NFS, there is no any *HyFS* daemon running on storage servers, and thus *HyFS* cannot fully take advantage of the resources of storage servers to perform any background operations, such as data scrubbing, or data migration. Due to the above reasons, we believe that a better approach is to implement a new network file system for *HyFS*. The new network file system does not have to be a general file system; rather, it could be tailored only for *HyFS*. We have to find what file services should be provided by this file system, so that *HyFS* can easily access and manage storage servers.

7.2.3 Scalability

The current *HyFS* can efficiently support up to 50 machines. This is a small environment. As a cloud storage system may contain thousands of machines, *HyFS* cannot work in this system with high performance. The problem lies in that the current *HyFS* uses all storage nodes to form a redundant group. Then, each codeword is stored across all nodes in the group. Although *HyFS* can use thousand machines for a group, the data write/read or reconstruction performance will be dramatically low because too many machines would be involved in serving one write or read request. There could be two solutions to improve the scalability of *HyFS*. One solution is to partition the machines to multiple separately groups. One user file is limited to be stored in one group. Because we can always keep a small number of machines in one group, *HyFS* will be

able to scale to a large amount of machines. The second approach is to have a configuration of stripe length. Stripe length means the number of consecutive stripes that are stored continuously at the same machines. The default stripe length of *HyFS* is unlimited, and thus a file is stored across a fixed number of machines in the system regardless of the machine number. With the configuration of stripe length, *HyFS* can distribute the data of a file on many machines rather than a fixed number of ones. For stripe width, we can use a small number, and then the write/read performance for a stripe can be independent from the machine number.

REFERENCES

[1] http://fuse.sourceforge.net/sshfs.html.

[2] http://www.tuxera.com/community/ntfs-3g-download/.

[3] http://www.iozone.org/.

[4] http://httpd.apache.org/docs/2.0/programs/ab.html.

[5] http://www.hpl.hp.com/research/linux/httperf/.

[6] http://sysbench.sourceforge.net/.

[7] Randy Allen and Ken Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann, October 2001.

[8] Diego F. Aranha. RELIC is an Efficient Library for Cryptography, version 0.2.3, 2010. http://code.google.com/p/relic-toolkit/.

[9] Diego F. Aranha, Julio López, and Darrel Hankerson. Efficient Software Implementation of Binary Field Arithmetic Using Vector Instruction Sets. In *LATINCRYPT '10: The First International Conference on Cryptology and Information Security in Latin America*, August 2010.

[10] Roberto Avanzi and Nicolas Thériault. Effects of Optimizations for Software Implementations of Small Binary Field Arithmetic. In *WAIFI '07: International Workshop on the Arithmetic of Finite Fields*, pages 21–22, September 2007.

[11] Daniel V. Bailey and Christof Paar. Optimal Extension Fields for Fast Arithmetic in Public-Key Algorithms. In *CRYPTO '98: Proc. of the Annual International Cryptology Conference*, 1998.

[12] Lakshmi Bairavasundaram, Garth R. Goodson, Bianca Schroeder, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. An Analysis of Data Corruption in the Storage Stack. In *FAST '08: Proc. of the 6th USENIX Conference on File and Storage Technologies*, February 2008.

[13] Lakshmi N. Bairavasundaram, Garth R. Goodson, Shankar Pasupathy, and Jiri Schindler. An Analysis of Latent Sector Errors in Disk Drives. In *SIGMETRICS '07: Proc. of ACM International Conference on Measurement and Modeling of Computer Systems*, June 2007.

[14] Mary Baker, Mehul Shah, David S. H. Rosenthal, Mema Roussopoulos, Petros Maniatis, TJ Giuli, and Prashanth Bungale. A Fresh Look at the Reliability of Long-term Digital Storage. In *EuroSys '06: 1st ACM SIGOPS/EuroSys European Conference on Computer Systems*, April 2006.

[15] Sandra Johnson Baylor, Peter Frank Corbett, and Chan ik Park. Efficient method for providing fault tolerance against double device failures in multiple device systems. *U. S. Patent 5,862,158*, January 1999.

[16] John A. Beachy and William D. Blair. *Abstract Algebra*. Waveland Press, Inc., 2006.

[17] Michael Ben-Or. Probabilistic Algorithms in Finite Fields. In *Symposium on Foundations of Computational Science*, pages 394–398, 1981.

[18] Anupam Bhide, Elmootazbellah N. Elnozahy, and Stephen P. Morgan. A Highly Available Network File Server. In *Proc. of the Winter 1991 USENIX Conference*, January 1991.

[19] Mario Blaum. A Family of MDS Array Codes with Minimal Number of Encoding Operations. In *ISIT '06: Proc. of IEEE International Symposium on Information Theory*, July 2006.

[20] 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, February 1995.

[21] Mario Blaum, Jehoshua Bruck, and Alexander Vardy. MDS Array Codes with Independent Parity Symbols. *IEEE Transactions on Information Theory*, 42(2), March 1996.

[22] Mario Blaum and Ron M. Roth. New Array Codes for Multiple Phased Burst Correction. *IEEE Transactions on Information Theory*, 39 (1):66–77, January 1993.

[23] Mario Blaum and Ron M. Roth. On Lowest Density MDS Codes. *IEEE Transactions on Information Theory*, 45(1):46–59, January 1999.

[24] Johannes Blomer, Malik Kalfane, Richard Karp, Marek Karpinski, Michael Luby, and David Zuckerman. An XOR-based Erasure-Resilient Coding Scheme. *Technical Report TR-95-048, International Computer Science Institute*, August 1995.

[25] Vasken Bohossian, Chenggong C. Fan, Paul S. LeMahieu, Marc D. Riedel, Lihao Xu, and Jehoshua Bruck. Computing in the RAIN: A Reliable Array of Independent Nodes. *IEEE Transaction on Parallel and Distributed Systems*, 12(2):99–114, 2001.

[26] Cachegrind. Cachegrind: a cache and branch-prediction profiler, 2010. http://valgrind.org/docs/manual/cg-manual.html.

[27] Philip H. Carns, Walter B. Ligon III, Robert B. Ross, and Rajeev Thakur. PVFS: A Parallel File System For Linux Clusters. In *Proc. of the 4th Annual Linux Showcase and Conference*, October 2000.

[28] Bruce Cassidy and James Lee Hafner. Space Efficient Matrix Methods for Lost Data Reconstruction in Erasure Codes. *IBM Research Report, rj10415*, September 2007.

[29] Feng Chen, David Koufaty, and Xiaodong Zhang. Understanding Intrinsic Characteristics and System Implications of Flash Memory based Solid State Drives. In *Proc. of ACM SIGMETRICS/Performance conference*, June 2009.

[30] Cleversafe Inc. Cleversafe Dispersed Storage, 2008. http://www.cleversafe.org/downloads.

[31] Peter Corbett, Bob English, Atul Goel, Tomislav Grcanac, Steven Kleiman, James Leong, and Sunitha Sankar. Row-Diagonal Parity for Double Disk Failure Correction. In *FAST '04: Proc. of the 3rd USENIX Conference on File and Storage Technologies*, March 2004.

[32] Whitfield Diffie and Martin E. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, 22 (6):644–654, November 1976.

[33] Cezary Dubnicki, Leszek Gryz, Lukasz Heldt, Michal Kaczmarczyk, Wojciech Kilian, Przemyslaw Strzelczak, Jerzy Szczepkowski, Cristian Ungureanu, and Michal Welnicki. HYDRAstor: A Scalable Secondary Storage. In *FAST '09: Proc. of the 7rd USENIX Conference on File and Storage Technologies*, February 2009.

[34] Jon G. Elerath and Michael Pecht. Enhanced Reliability Modeling of RAID Storage Systems. In *DSN '07: International Conference on Dependable Systems and Networks*, June 2007.

[35] Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472, 1985.

[36] National Institute for Standards and Technology. FIPS 186-3: Digital Signature Standard (DSS), 2009. http://www.itl.nist.gov/fipspubs/by-num.htm.

[37] Robert G. Gallager. *Low-Density Parity-Check Codes*. M.I.T. Press, Cambridge, MA, 1963.

[38] Shuhong Gao and Daniel Panario. Tests and constructions of irreducible polynomials over finite fields. In *FoCM'97: Foundations of Computational Mathematics*, 1997.

[39] gentoo wiki, 2010. http://en.gentoo-wiki.com/wiki/CFLAGS.

[40] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *SOSP '03: Proc. of the 19th ACM Symposium on Operating Systems Principles*, 2003.

[41] Garth A. Gibson, Daniel Stodolsky, Fay W. Chang, William V. Courtright, Chris G. Demetriou, Eka Ginting, Mark Holland, Qingming Ma, LeAnn Neal, R. Hugo Patterson, Jiawen Su, Rachad Youssef, and Jim Zelenka. The Scotch Parallel Storage Systems. In *COMPCON '95: Proc. of 40th IEEE Computer Society International Conference*, March 1995.

[42] Edgar Nelson Gilbert. Capacity of a Burst-Noise Channel. *Bell System Technical Journal*, 39:1253–1266, September 1960.

[43] Gluster Inc., 2011. http://www.gluster.org/.

[44] Kevin M. Greenan, Ethan L. Miller, and Thomas J. E. Schwarz. Analysis and Construction of Galois Fields for Efficient Storage Reliability. In *Technical Report UCSC-SSRC-07-09*, August 2007.

[45] Kevin M. Greenan, Ethan L. Miller, and Thomas J. E. Schwarz. Optimizing Galois Field Arithmetic for Diverse Processor Architectures and Applications. In *MASCOTS '08: International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, September 2008.

[46] Jorge Guajardo, Sandeep S. Kumar, Christof Paar, and Jan Pelzl. Efficient Software-Implementation of Finite Fields with Applications to Cryptography. *Acta Applicandae Mathematicae*, 93:3–32, September 2006.

[47] James Lee Hafner. WEAVER Codes: Highly Fault Tolerant Erasure Codes for Storage Systems. In *FAST '05: Proc. of the 4th USENIX Conference on File and Storage Technologies*, December 2005.

[48] James Lee Hafner, Veera Deenadhayalan, Wendy Belluomini, and KK Rao. Undetected Disk Errors in RAID Arrays. *IBM Journal of Research and Development*, 52(4/5):413–425, July/September 2008.

[49] James Lee Hafner, Veera Deenadhayalan, KK Rao, and John A. Tomlin. Matrix Methods for Lost Data Reconstruction in Erasure Codes. In *FAST '05: Proc. of the 4th USENIX Conference on File and Storage Technologies*, December 2005.

[50] Darrel Hankerson, Julio López Hernandez, and Alfred Menezes. Software Implementation of Elliptic Curve Cryptography Over Binary Fields. In *CHES '00: Workshop on Cryptographic Hardware and Embedded Systems*, 2000.

[51] Greg Harper, Alfred Menezes, and Scott Vanstone. Public-Key Cryptosystems with Very Small Key Lengths. In *Eurocrypt '92: Proc. of the Annual International Conference on the Theory and Applications of Cryptographic Techniques*, 1992.

[52] John H. Hartman and John K. Ousterhout. The Zebra Striped Network File System. *ACM Transactions on Computer Systems*, 13:274–310, August 1995.

[53] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*, chapter Appendiex C. Morgan Kaufmann, May 2002.

[54] Chen Huang and Lihao Xu. STAR: An Efficient Coding Scheme for Correcting Triple Storage Node Failures. In *FAST '05: Proc. of the 4th USENIX Conference on File and Storage Technologies*, December 2005.

[55] Cheng Huang, Jin Li, and Minghua Chen. On Optimizing XOR-Based Codes for Fault-Tolerant Storage Applications. In *ITW '07: Proc. of IEEE Information Theory Workshop*, September 2007.

[56] Cheng Huang and Lihao Xu. Fast Software Implementation of Finite Field Operations. *Technical report, Washington University*, 2003.

[57] Cheng Huang and Lihao Xu. Study of A Practical FEC Scheme for Wireless Data Streaming. In *IASTED '05: Proc. of Internet and Multimedia Systems and Applications*, February 2005.

[58] Cluster File Systems Inc. A Scalable, High-performance File System. *http://www.lustre.org*, 2006.

[59] Intel. Intel SSE4 Programming Reference, 2007. http://software.intel.com/file/18187/.

[60] Intel. Intel 64 and IA-32 Architectures Software Developer's Manuals, Volumn 1, 2010. http://www.intel.com/Assets/PDF/manual/253666.pdf.

[61] Intel. Intel 64 and IA-32 Architectures Software Developer's Manuals, Volumn 3B, 2010. http://www.intel.com/Assets/PDF/manual/253669.pdf.

[62] Intel. Intel Advanced Encryption Standard (AES) Instructions Set, 2011. http://software.intel.com/file/24917.

[63] Intel VTune, 2010. http://software.intel.com/en-us/intel-vtune/.

[64] Larry Jones, Matt Reid, Marc Unangst, Garth Gibson, and Brent Welch. Panasas Tiered Parity Architecture. *Panasas white paper: http://storage-brain.com/wp-content/uploads/papers/Tiered_Parirty_Arch-May2010.pdf*, May 2011.

[65] Nikolai Joukov, Arun M. Krishnakumar, Chaitanya Patti, Abhishek Rai, Sunil Satnur, Avishay Traeger, and Erez Zadok. RAIF: Redundant Array of Independent Filesystems. In *MSST '07: Proc. of the 24th IEEE Conference on Mass Storage Systems and Technologies*, pages 199–212, September 2007.

[66] David W. Kravitz. Digital signature algorithm, 1993. U.S. Patent 5,231,668.

[67] Andrew Krioukov, Lakshmi Bairavasundaram, Garth R. Goodson, Kiran Srinivasan, Randy Thelen, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Parity Lost and Parity Regained. In *FAST '08: Proc. of the 6th USENIX Conference on File and Storage Technologies*, February 2008.

[68] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao. OceanStore: An Architecture for Global-scale Persistent Storage. In *ASPLOS '00: Proc. of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, December 2000.

[69] Alvin R. Lebeck and David A. Wood. Cache Profiling and the SPEC Benchmarks: A Case Study. *IEEE Computer*, 27 (10):15–26, 1994.

[70] Adam Leventhal. Triple-Parity RAID and Beyond. http://queue.acm.org/detail.cfm?id=1670144.

[71] Rudolf Lidl and Harald Niederreiter. *Finite Fields*. Cambridge University Press, 1997.

[72] Barbara Liskov, Sanjay Ghemawat, Robert Gruber, Paul Johnson, Liuba Shrira, and Michael Williams. Replication in the Harp File System. In *SOSP '91: Proc. of the 13th ACM Symposium on Operating Systems Principles*, October 1991.

[73] Julio López and Ricardo Dahab. High-speed Software Multiplication in $\mathbb{F}_{2^m}$. In *INDOCRYPT '00: Proc. of the Annual International Conference on Cryptology in India*, 2000.

[74] Jiwei Lu, Howard Chen, Rao Fu, Wei-Chung Hsu, Bobbie Othmer, Pen-Chung Yew, and Dong-Yuan Chen. The Performance of Runtime Data Cache Prefetching in a Dynamic Optimization System. In *MICRO'03: 36th Annual International Symposium on Microarchitecture*, December 2003.

[75] Michael Luby. Code for Cauchy Reed-Solomon Coding, 1997. http://www.icsi.berkeley.edu/~luby/cauchy.tar.uu.

[76] Jianqiang Luo, Cheng Huang, and Lihao Xu. Decoding star code for tolerating simultaneous disk failure and silent errors. In *DSN '10: The International Conference on Dependable Systems and Networks*, June 2010.

[77] Jianqiang Luo, Lihao Xu, and James S. Plank. An efficient XOR-Scheduling algorithm for erasure codes encoding. In *DSN '09: The International Conference on Dependable Systems and Networks*, June 2009.

[78] Lustre File System. High-Performance Storage Architecture and Scalable Cluster File System. *Lustre white paper: http://wiki.lustre.org/index.php/Lustre_Publications*, December 2007.

[79] Florence Jessie MacWilliams and Neil J. A. Sloane. *The Theory of Error Correcting Codes*. Amsterdam: North-Holland, 1977.

[80] Florence Jessie MacWilliams and Neil J. A. Sloane. The Theory of Error-Correcting Codes. *Northholland, Amsterdam, The Netherlands*, February 1977.

[81] Alfred Menezes, Paul van Oorschot, and Scott Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997.

[82] Victor S. Miller. Use of Elliptic Curves in Cryptography. In *CRYPTO 85': Proc. of the Annual International Cryptology Conference*, 1986.

[83] Todd C. Mowry, Monica S. Lam, and Anoop Gupta. Design and Evaluation of a Compiler Algorithm for Prefetching. In *ASPLOS'92: 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1992.

[84] Brad Nisbet. FAS storage systems: Laying the foundation for application availability. *Network Appliance white paper: http://www.netapp.com/us/library/analyst-reports/ar1056.html*, February 2008.

[85] David A. Patterson, Garth Gibson, and Randy H. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *SIGMOD '88: Proc. of the 1988 ACM SIGMOD International Conference on Management of Data*, 1988.

[86] Eduardo Pinheiro, Wolf-Dietrich Weber, and Luiz Andre Barroso. Failure Trends in a Large Disk Drive Population. In *FAST '07: Proc. of the 5th USENIX Conference on File and Storage Technologies*, February 2007.

[87] James. S. Plank. A Tutorial on Reed-Solomon Coding for Fault Tolerance in RAID-like Systems. *Software – Practice and Experience*, 27 (9):995–1012, September 1997.

[88] James. S. Plank. Fast Galois Field Arithmetic Library in C/C++, 2007. http://www.cs.utk.edu/~plank/plank/papers/CS-07-593/.

[89] James S. Plank. Jerasure: A Library in C/C++ Facilitating Erasure Coding for Storage Applications. *Tech. Rep. CS-07-603, University of Tennessee*, September 2007.

[90] James S. Plank. The RAID-6 Liberation Codes. In *FAST '08: Proc. of the 6th Usenix Conference on File and Storage Technologies*, February 2008.

[91] James S. Plank, Adam L. Buchsbaum, Rebecca L. Collins, and Michael. G. Thomason. Small Parity-Check Erasure Codes - Exploration and Observations. In *DSN '05: Proc. of the International Conference on Dependable Systems and Networks*, June 2005.

[92] James S. Plank, Jianqiang Luo, Catherine D. Schuman, Lihao Xu, and Zooko Wilcox-O'Hearn. A Performance Evaluation and Examination of Open-Source Erasure Coding Libraries For Storage. In *FAST '09: Proc. of the 7th Usenix Conference on File and Storage Technologies*, February 2009.

[93] James. S. Plank, Scott Simmerman, and Catherine D. Schuman. Jerasure: A Library in C/C++ Facilitating Erasure Coding for Storage Applications. Technical Report CS-08-627, University of Tennessee, August 2008.

[94] James S. Plank and Lihao Xu. Optimizing Cauchy Reed Solomon Codes for Fault-Tolerant Network Storage Applications. In *NCA '06: Proc. of the 5th IEEE International Symposium on Network Computing and Applications*, July 2006.

[95] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. ArpaciDusseau, , and Remzi H. ArpaciDusseau. IRON File Systems. In *SOSP '05: Proc. of the 20th ACM Symposium on Operating Systems Principles*, October 2005.

[96] Irving S. Reed and Gustave Solomon. Polynomial Codes over Certain Finite Fields. *Journal of the Society for Industrial and Applied Mathematics*, 8 (10):300–304, 1960.

[97] Mahadev Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, and David C. Steere. Coda: A Highly Available File System for a Distributed Workstation Environment. *IEEE Transactions on Computers*, 39, April 1990.

[98] Bianca Schroeder and Garth A. Gibson. Disk Failures in the Real World: What Does an MTTF of 1,000,000 Hours Mean to You? In *FAST '07: Proc. of the 5th USENIX Conference on File and Storage Technologies*, February 2007.

[99] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. DRAM errors in the wild: A Large-Scale Field Study. In *Proc. of ACM SIGMETRICS/Performance conference*, June 2009.

[100] Richard Schroeppel, Hilarie Orman, Sean O' Malley, and Oliver Spatscheck. Fast Key Exchange with Elliptic Curve Systems. In *CRYPTO '95: Proc. of the Annual International Cryptology Conference*, 1995.

[101] Thomas J. Schwarz, Qin Xin, Ethan L. Miller, Darrell D. Long, Andy Hospodor, and Spencer Ng. Disk Scrubbing in Large Archival Storage Systems. In *MASCOTS '04: Proc. of the 12th Annual Meeting of the IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, October 2004.

[102] Gadiel Seroussi. Table of Low-Weight Binary Irreducible Polynomials, 1998. http://www.hpl.hp.com/techreports/98/HPL-98-135.pdf.

[103] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *MSST '10: Proc. of the 26th IEEE Symposium on Massive Storage Systems and Technologies*, 2010.

[104] Miklos Szered. FUSE: Filesystem in Userspace. *http://fuse.sourceforge.net*, 2007.

[105] R. Michael Tanner. A recursive approach to low complexity codes. *IEEE Transactions on Information Theory*, 27 (5):533–547, September 1981.

[106] Cristian Ungureanu, Benjamin Atkin, Akshat Aranya, Salil Gokhale, Stephen Rago, Grzegorz Calkowski, Cezary Dubnicki, and Aniruddha Bohra. HydraFS: A High-Throughput File System for the HYDRAstor Content-Addressable Storage System. In *FAST '10: Proc. of the 8th USENIX Conference on File and Storage Technologies*, Feburary 2010.

[107] Brigitte Vallée. The Complete Analysis of the Binary Euclidean Algorithm. In *ANTS '98: Proc. of the Third International Symposium on Algorithmic Number Theory Symposium*, 1998.

[108] Victor Shoup. A New Polynomial Factorization Algorithm and Its Implementation. *Journal of Symbolic Computation*, 20:363–397, 1996.

[109] Joe Warren. A Hierarchical Basis for Reordering Transformations. In *POPL '84: Proc. of the 11th ACM SIGACT-SIGPLAN symposium on Principles of Programming Languages*, January 1984.

[110] Brent Welch, Marc Unangst, Zainul Abbasi, Garth Gibson, Brian Mueller, Jason Small, Jim Zelenka, and Bin Zhou. Scalable Performance of the Panasas Parallel File System. In *FAST '08: Proc. of the 6th USENIX Conference on File and Storage Technologies*, February 2008.

[111] Stephen B. Wicker. *Error Control Systems for Digital Communication and Storage*. Prentice Hall, 1994.

[112] Alden Wilner. Multiple Drive Failure Tolerant RAID System. *U. S. Patent 6,327,672*, December 2001.

[113] Erik De Win, Antoon Bosselaers, Servaas Vanderberghe, Peter De Gersem, and Joos Vandewalle. A Fast Software Implementation for Arithmetic Operations in $GF(2^n)$. In *ASIACRYPT '96: Proc. of the Annual International Conference on the Theory and Application of Cryptology Information Security*, 1996.

[114] Jay J. Wylie and Ram Swaminathan. Determining Fault Tolerance of XOR-Based Erasure Codes Efficiently. In *DSN '07: Proc. of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, June 2007.

[115] Lihao Xu. X-Code: MDS Array Codes with Optimal Encoding. *IEEE Transactions on Information Theory*, 45 (1):272–276, January 1999.

[116] Lihao Xu. Hydra: A Platform for Survivable and Secure Data Storage Systems. In *StorageSS '05: International Workshop on Storage Security and Survivability*, November 2005.

[117] Lihao Xu, Vasken Bohossian, Jehoshua Bruck, and David G. Wagner. Low Density MDS Code and Factors of Complete Graphs. *IEEE Transactions on Information Theory*, 45:1817–1826, 1999.

[118] Erez Zadok, Rakesh Iyer, Nikolai Joukov, Gopalan Sivathanu, and Charles P. Wright. On Incremental File System Development. *ACM Transactions On Computer Systems*, 2 (2), May 2006.

[119] G. V. Zaitsev, V. A. Zinovev, and N. V. Semakov. Minimum-check-density codes for correcting bytes of errors. *Problems of Information Transmission*, 19 (3):197–204, 1983.

ABSTRACT

HYFS: DESIGN AND IMPLEMENTATION OF A RELIABLE FILE SYSTEM

by

JIANQIANG LUO

AUGUST 2011

Advisor: Dr. Lihao Xu

Major: Computer Science

Degree: Doctor of Philosophy

Building reliable data storage systems is crucial to any commercial or scientific applications. Modern storage systems are complicated, and they are comprised of many components, from hardware to software. Problems may occur to any component of storage systems and cause data loss. When this kind of failures happens, storage systems cannot continue their data services, which may result in large revenue loss or even catastrophe to enterprises. Therefore, it is critically important to build reliable storage systems to ensure data reliability.

In this dissertation, we propose to employ general erasure codes to build a reliable file system, called *HyFS*. *HyFS* is a cluster system, which can aggregate distributed storage servers to provide reliable data service. On client side, *HyFS* is implemented as a native file system so that applications can transparently run on top of *HyFS*. On server side, *HyFS* utilizes multiple distributed storage servers to provide highly reliable data service by employing erasure codes. *HyFS* is able to offer high throughput for either random or sequential file access, which makes *HyFS* an attractive choice for primary or backup storage systems.

This dissertation studies five relevant topics of *HyFS*. Firstly, it presents several algorithms that can perform encoding operation efficiently for XOR-based erasure codes. Secondly, it discusses an efficient decoding algorithm for RAID-6 erasure codes. This algorithm can recover various types of disk failures. Thirdly, it describes an efficient algorithm to detect and correct errors for the STAR code, which further improves a storage system's reliability. Fourthly, it describes efficient implementations for the arithmetic operations of large finite fields. This is to improve a storage system's security. Lastly and most importantly, it presents the design and implementation of *HyFS* and evaluates the performance of *HyFS*.

AUTOBIOGRAPHICAL STATEMENT

Jianqiang Luo received M.S. degree of Computer Application Technology from Shanghai Jiao Tong University, Shanghai, China, in 2004. During the Ph.D program, he worked in Network and Information Systems Lab (NISL) in the Department of Computer Science, Wayne State University, Detroit, Michigan. His research interests are in the areas of error correcting codes, reliable storage systems, and large scale distributed systems.