Lehigh University
Lehigh Preserve

Theses and Dissertations

2011

Coding for storage: disk arrays, flash memory, and distributed storage networks

Nattakan Puttarak
Lehigh University

Follow this and additional works at: http://preserve.lehigh.edu/etd

Recommended Citation
Puttarak, Nattakan, "Coding for storage: disk arrays, flash memory, and distributed storage networks" (2011). *Theses and Dissertations.* Paper 1144.

This Dissertation is brought to you for free and open access by Lehigh Preserve. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of Lehigh Preserve. For more information, please contact preserve@lehigh.edu.

CODING FOR STORAGE: DISK ARRAYS, FLASH MEMORY AND DISTRIBUTED STORAGE NETWORKS

by

Nattakan Puttarak

A Dissertation
Presented to the Graduate Committee
of Lehigh University
in Candidacy for the Degree of
Doctor of Philosophy
in
Electrical Engineering

Lehigh University
September 2011

© Copyright 2011 by Nattakan Puttarak
All Rights Reserved

This dissertation is accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

____
(Date)

____
Professor Tiffany Jing Li
(Dissertation Advisor)

____
(Accepted Date)

____
Professor Tiffany Jing Li
(Dissertation Advisor)

____
Professor Shalinee Kishore

____
Professor Meghanad D. Wagh

____
Professor Liang Cheng
(Department of Computer Science and Engineering)

To my parents, my sister, my brother-in-law, and my niece.

Acknowledgements

This PhD work would not have been completed without a great deal of support and guidance from a number of people. In order to show my gratitude towards these people, I would like to dedicate this page to them.

First of all, I would like to deeply thank the most important person that made this dissertation possible, my advisor, Professor Tiffany Jing Li, for giving me the opportunity to join this research group. With her constant guidance, expertise, energy and inspiration, she has been my best mentor, and advisor. I have developed not only my technical skills, attitudes and knowledge, but also unconsciously learned how to attain an optimistic perspective of life from her. This work would not have been possible without her.

I would also like to express my gratitude to the rest of my committee members: Professor Shalinee Kishore, Professor Meghanad D. Wagh and Professor Liang Cheng who have provided valuable feedback, direction and support in my research.

I would like to thank Thai Government and the King Mongkut's Institute of Technology Lardkrabang (KMITL) for the scholarship which has supported me throughout the entire graduate program in US.

I would not have gone through my PhD experience without the constant interaction with my fellow lab colleagues and friends. Many thanks to Dr. Peiyu Tan, Dr. Xingkai Bao, Dr. Kai Xie, Phisan Kaewprapha, Dr. Vitchanetra

Hongpinyo, and Yang Liu for the invaluable discussion, great support, unconditional help, and friendship.

Last but not least, I would like to dedicate this dissertation to my family. I am so grateful to my parents, sister, brother-in-law and my niece whom I could not have asked for anything more, for their support and encouragement and for always being there for me when I am facing hardships. I will not be the person I am today without them.

Contents

Acknowledgements **vii**

Abstract **1**

1 Introduction **3**

1.1 Disk Drives and The Distributed Data Storages 6

1.1.1 Notations and Definitions of Disk Arrays 8

1.1.2 Backgrounds of the RAID levels and erasure-correcting codes 9

1.2 Coding Theory for Data Storages 13

1.2.1 Reed-Solomon (RS) Codes 15

1.2.2 LDPC Codes 17

1.2.3 Parity Array Codes 19

1.3 Flash Drives 25

1.3.1 NOR vs. NAND Flash Memory 27

1.4 Outline 30

2 MDS codes for disk arrays **33**

2.1 Introduction 33

2.1.1 MDS Codes and Their Properties 34

2.1.2 Literature Reviews 35

2.2 CGR Codes 37

2.2.1 Code Construction and Algorithms 37

2.3 Proofs of CGR Array Codes 44
2.3.1 Proofs of an MDS Property of CGR Codes 44
2.3.2 Perfect One-Factorization (P1F) as the Inter-Ring Edges Shifting Index Assigning Algorithm 52
2.4 Dual CGR Codes 58
2.4.1 Proofs of Duality of CGR Codes 59
2.5 Connection to B-Codes 63
2.5.1 Discussion 67
2.6 Low-Density MDS Array Codes 69
2.6.1 Low-Density CGR Codes 71
2.6.2 Data Recovery via Parity-Check Matrix 76

3 Nested codes with Hierarchical protection for distributed storage networks 81
3.1 Introduction 82
3.1.1 Background of Luby Transform (LT) Codes 84
3.2 The MDS-LT Nested Codes 87
3.2.1 The Code Construction 88
3.2.2 The Consideration of Hierarchical Nested Erasure Codes 92
3.3 The Horizontal-Vertical Single Parity Check (HVSPC) Codes . 99
3.3.1 Simulation Results and Analysis 101
3.4 Summary 104

4 Coding for flash memory 107
4.1 Introduction 107
4.1.1 How Flash Memories work? 108
4.1.2 Literature Reviews 110
4.1.3 The Number of Writes Consideration 115
4.2 The Word-write Efficient and Bit-write Efficient (WEBE) Codes 116
4.2.1 Problem Formulation and New Concepts 118
4.2.2 Design WEBE Codes for $k = 2$ 122

4.2.3 Design WEBE Codes for General k 130
4.3 Flash Marker (FM) Codes 137
4.3.1 FM Code Construction 139
4.3.2 Simulation Results 145
4.3.3 Discussion 148
4.4 Conclusion 150

5 Summary and Future Works **151**
5.1 Data Disks 152
5.2 The Distributed Storage Networks 152
5.3 Flash Memories 153

Bibliography **155**

List of Tables

1.1 Strengths and weaknesses of standard RAID levels 14
1.2 An example of a simple EVENODD code 20
1.3 An (5×5) array of X-code 21
1.4 An (4×6) array of RDP code 22
1.5 An (4×8) array of STAR code 23
1.6 Erasure codes for disk storage arrays 25
1.7 The properties and performances of NOR and NAND flash memories 30

2.1 B_0 50
2.2 B_1 51
2.3 B_2 52
2.4 B_3 53
2.5 B_4 54
2.6 B_5 55
2.7 B_6 56
2.8 Update complexity and decoding complexity 69
2.9 The array $\mathrm{CGR}(K_2, C_5)$ code 73

List of Figures

1.1 The wireless communication system model 4

1.2 The data read/write model 5

1.3 The illustration of terminology in an horizontal erasure code . 8

1.4 Optional caption for list of figures 11

1.5 The Reed Solomon (RS) codes for disk arrays 16

1.6 An example of a simple LDPC code with $n = 3, m = 2$ 17

1.7 The $HoVer^{t}_{v,h}[r, c]$ codes. 24

1.8 MOS memory tree 26

2.1 CGR graphs constructed from base graphs. Left: base graphs K_2 and K_4; right: resultant CGR graphs $CGR(K_2, C_5)$ and $CGR(K_4, C_7)$. 38

2.2 Labeling of 3-regular CGR(K_2,C_5). 40

2.3 Complete graph K_6. 43

2.4 A ring of complete graph of (K_4, C_7) 48

2.5 A Hamiltonian cycle formed by 2 survivors of (K_4, C_7) 57

2.6 Complete graph K_4 after trimming K_6 57

2.7 Graph representing a row in H 62

2.8 Graph representing a row in H of the dual code 63

2.9 A super graph represents a CGR(K_4, C_7) code, where each super node has 7 nodes and there are 7 edges represented in each inter-edge. 64

2.10 (a) Structure of CGR code. (b) Structure of B_{2n+1} code . . . 68

2.11 Optional caption for list of figures 74
2.12 A row-decoding process of the H matrix of $CGR(K_2, C_5)$ code 78
2.13 A column-decoding process of the H matrix of $CGR(K_2, C_5)$ code 79

3.1 Two types of stripe layouts of GRID(SPC,EVENODD) codes . 83
3.2 The decoding process when there are $u - 1$ input symbols are undecoded 86
3.3 The basic structure of nested codes with Hierarchical protection for distributed storage networks 89
3.4 Code array structure where M global disks are all parity disks constructed from LT codes 92
3.5 The probability of residual disk errors versus the raw disk failure rate (P_e). 93
3.6 The ability of the hierarchy nested erasure code to recover failed disks in time period t. 94
3.7 Comparisons the probability of disk errors between Grid codes and hierarchy nested erasure codes 95
3.8 The EXIT chart of LT codes and MDS codes 97
3.9 The array structure 100
3.10 The organization of MDS local code in the array of size $x \times y$ 100
3.11 The probability of disk failures after applying the layered coding scheme. 101
3.12 The probability (in log-scale) of disk failures after applying the layered coding scheme. 102
3.13 The comparison of HVSPC codes and GRID(STAR,STAR) codes 103

4.1 Schematic cross section of flash memory. 109
4.2 A $(3, 2)_2$ flash code that achieves the maximum word-write efficiency 2. 121

4.3 A $(3,2)_2$ flash code (floating code in [46]) that achieves the maximum bit-write efficiency, but not the maximum word-write efficiency. 122
4.4 Relation between bit-write optimality and word-write optimality. 123
4.5 The proposed $(3,2)_q$ flash code. 127
4.6 An example of a simple $(6,3)_q$ WEBE code 134
4.7 A $(6,3)_2$ WEBE code that achieve an asymptotically optimal word-writes 134
4.8 One example of layout structures of $(n,k)_q$ WEBE code 138
4.9 The number of word-writes $(5,3)_q$ and $(6,3)_q$ WEBE codes for the various value of q. 138
4.10 The relation of s marker states, s spare cells of $(N,K,s)_q$ FM code 142
4.11 An example of cell-state updates of $(15,4,1)_4$ FM code shown in *Example 6* (all cells shown in the parentheses are spare cells). 146
4.12 The number of bit-writes of $(N,K,s)_q$ FM codes when the number of spare-cell units (s) is increased 147
4.13 The number of word-writes of $(N,K,s)_q$ FM codes when the number of spare-cell units (s) is increased. 149

Abstract

The explosive demand for digital data storage with higher areal density, larger storage capacity, higher reliability and fault tolerance, easier accessibility, cheaper management and better scalability, poses tremendous challenge on the storage industry. Researchers and practitioners have been working hard to tackle the problem in various aspects from system architecture to signal processing, coding, control and storage media. This doctoral research explores emerging coding technologies that will potentially lead to new and better storage systems to meet some of the above demanding goals. In this dissertation, we consider three important storage systems: hard disk arrays consisting of few disks, distributed storage networks consisting of hundreds of and thousands of disks, and flash memories. However, the coding for disk storages and the one for flash memories are different in terms of purposes, functionality, and technology.

In the case of disk arrays, we propose to develop new erasure codes to achieve optimal spatial efficiency while requiring only minimal encoding and decoding complexity. Specifically, we demonstrate the idea of constructing class of nested graphs, termed *complete-graph-of-ring (CGR) graphs*, and use them to form a class of optimal array codes, termed *CGR codes*. CGR codes are maximum distance separable (MDS), and hence achieve the best space efficiency. Systematic and concrete constructing methods for CGR codes and

their dual codes are developed. It is shown that these codes not only deliver optimal erasure protection with low complexity, but they also provide a rich array of code rates and code lengths, many of which are suitable for storage systems. The MDS array codes are also presented as the systematic low-density (sparse) array codes shown by the generator matrix and parity-check matrix.

For large distributed storage networks, we propose to develop layered coding strategies to achieve good erasure protection, without causing unbearable communication overhead. By dividing the entire system in layered clusters and designing appropriate erasure coding for each layer, we show that a good trade-off between protection capability, redundancy overhead, communication overhead, and computational complexity can be achieved. Additionally, the proposed strategy also provides the flexibility and scalability much need for large systems.

In the case of flash memories, we propose to develop new coding schemes to best map cell states to data bits and vice versa. Our goal is to maximize the writing time in each cell state before a block-erased process is required. The existing strategies can at the best achieve the "conventional bound" under the assumption that any one bit update will inevitably cause a cell state rises. We demonstrate an idea which allows some two-bit updates to be represented by only one cell state rises (rather than two cell states rise), a direction that people have not thought before. We also introduce the concept of word-write efficiency and optimality, and propose new classes of "*word-efficient bit-efficient (WEBE) codes*" and "*word-optimal bit-optimal (WOBO) codes*." To achieve flexibility and adaptivity, and further improve the lifespan of flash memories, we introduce the "*flash marker (FM) codes*," which reserve a set of cells for the most active bits in order to avoid a block erasure. From all of the above, we have beaten the conventional performance bound and opened new possibilities for data representation in flash memories.

Chapter 1

Introduction

We live in a "YouTube" age, in which an enormous amount of digital information is created every day. The explosive surge of data poses a serious demand for cheaper, better, and more reliable data storage that is portable (e.g. flash memory) and/or accessible anywhere and anytime (e.g. storage networks). Today's data storage industry is undergoing a paradigm shift, from a single prevailing media (e.g. magnetic hard disks) to a rich variety of media (e.d. magnetic hard disks, CD, DVD, and solid state storage), and from individual disks or small arrays of disks to very large storage networks comprising hundreds or thousands of (distributed) storage nodes. What this implies in research is the need to invent new storage technologies and improve existing ones.

The demand for massive storage comes with not just the requirement for a high storage capacity, but also for a high density (i.e. small space), fast accessibility, better reliability and fault tolerance, easy management, and good scalability. In the end, the efficiency of the storage technology is also measured by the per-unit (dollar) cost to store and maintain digital data, and every

effort is made to minimize this cost while maintaining a high availability and reliability of the system.

Compared to the wireless communication system model as shown in Fig. 1.1, instead of transmitting encoded symbols/information from source to receiver via various communication channels, in case of data storage, we read/write (store) information in the same disk for the numerous times. As shown in Fig. 1.2, the error on a storage device might be sporadic or bursty. In the latter case, the error source may be the classical scratch, the error from read/write failure, or controller [2].

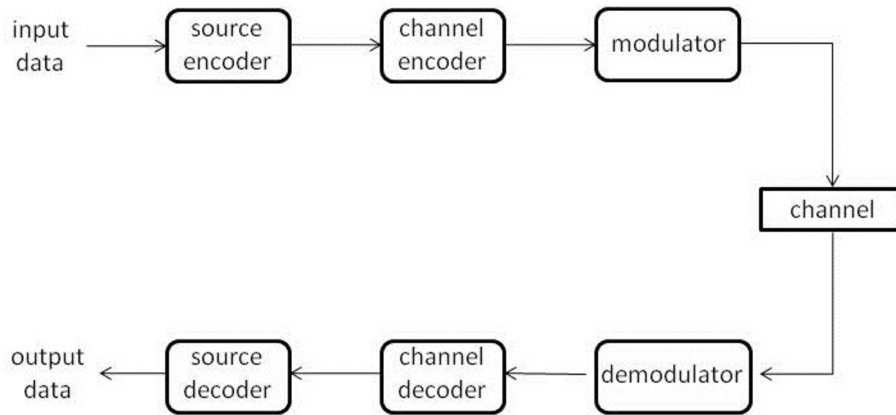


Figure 1.1: The wireless communication system model

This dissertation is centered around two types of storage devices: hard disk drives, which are and will remain the dominant large-volume data storage devices for the foreseeable near future, and flash drives (or flash memories), which are the dominant portable storage device that is gaining an increasingly large market for small to medium volumes. The "Coding Theory" is commonly called upon to improve and achieve the ability to efficiently store, access and

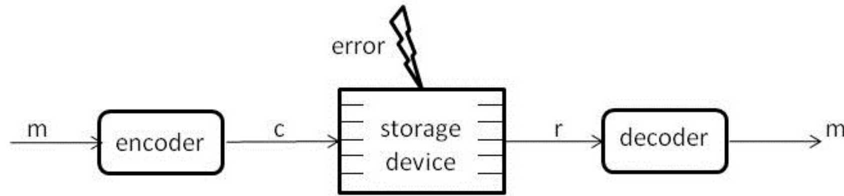


Figure 1.2: The data read/write model

transfer information in data disks and flash memories in a reliable way.

Since today's massive data can not be handled by only a single hard disk, but rather must rely on the collection of multiple disks, we consider two levels of hard disk collections: in the small scale, we consider disk arrays, and in the large scale, we consider data centers (or distributed data storage systems) consisting of hundreds of or thousands of storage nodes, each of which comprising an array of disks. To achieve reliable and fast-recovery data storage that is essential to support data availability, persistence, and integrity, we exploit advanced *erasure coding* technology. Array codes– a class of linear erasure codes– play an important role in storage systems, due to their simplicity such that the encoding and decoding procedures are performed only by exclusive-OR (XOR) operations. Specifically, (array) codes that achieve the maximal spacial efficiency or the Singleton bound are called maximum distance separable (MDS) (array) codes. We propose to search for new directions and new ways to construct MDS array codes. We will look specifically into constructions relating to graph and set theory. Our research goal is to achieve MDS with rich choices of code lengths and rates, and with minimal encoding and decoding complexity possible.

Flash drives are a young technology but potentially very promising. They

have desirable properties including high data density, fast reading time, physical robustness (can withstand drops) and small sizes. Hence, they have found wide application in portable devices such as MP3 player, mobile phones, digital cameras, or computer laptops. Compared to hard disks and optical disks where the media provide two distinctive states to represent (i.e. store) 0s and 1s, flash memories have many levels of cell states that can be used to represent digital data. The state can be easily increased by injecting an electron into the cell level, but to decrease the cell state level one must erase the entire block and reprogram all the cells, a procedure called *black erasure* which is both costly and slow [45]. Hence, in order to achieve the full efficiency of flash memories, the proposed research targets developing strategies to maximize the limited life cycle of flash memories, namely the life span, or to maximize the number of writing before the erasing process is needed. Here we will investigate new ideas and ways to efficiently map information bits into cell states and to represent the writing levels when a charge is added (written) into a flash memory.

1.1 Disk Drives and The Distributed Data Storages

Since a huge amount of information is stored and transferred among many storage and data centers, data loss due to disk failure (i.e. erasure) is a major issue that may affect the reliability of this system. Reliability and performance of storage systems are a big concern, and are an important aspect of the reliability and performance of the overall cyber infrastructure. A recent trend in storage is that, instead of using a very expensive, high performance, and large capacity disk storage to store voluminous data, a group of several cheaper, low-density and lower capacity disks are combined into one logical unit called

a "disk array". Disk arrays provide a cost-effective means to mitigate the problem of data loss, since they contain multiple redundant disk drives to address the fault tolerance. There are several key aspects in this multiple disk storage mechanism as mentioned in the following.

- Reliability: fault tolerance and robustness, which must be built into the system to recover/tolerate disk failures. If there is a failure, the system is not reliable.
- Availability: the ability for the system to work in times of individual disk failure. When a system can continue to work even in the presence of a failure of one or more disks, the system is called to be available.
- Scalability: the ability to gracefully support a system when a data center grows in size or when two data centers merge.
- Flexibility: the ability to be arranged or configured in different ways to satisfy different system requirements.
- Capacity: the ability to handle thousand of disks within the same network to collectively provide massive storage capability.
- Speed: time efficiency in accessing required information. The faster the access speed, the better the delay time should be minimized as much as possible.

The well-known disk arrays used in industry are the Reliable/Redundant Arrays of Inexpensive/Independent Disks (RAID) system was proposed by Patterson, et al. in 1988 [17]. RAID systems can offer fault tolerance and a higher throughput level than a single hard drive. In addition, RAID provides a combination of outstanding data availability, highly scalable performance, high capacity, and recovery.

1.1.1 Notations and Definitions of Disk Arrays

For clarity and consistency of the use of common terms in storage and erasure codes, we express and state all the definitions here to avoid confusion. Following the convention in [9], [11], and [39], the terminology used throughout this dissertation is represented in Fig. 1.3, which shows a horizontal erasure code.

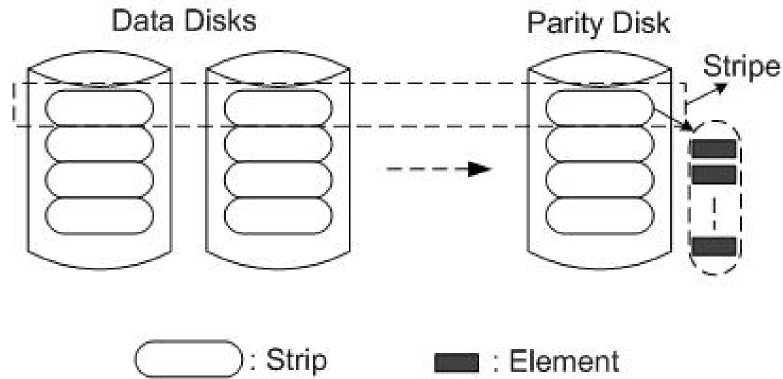


Figure 1.3: The illustration of terminology in an horizontal erasure code

Data is a chunk of bytes or blocks containing unmodified user data, while *parity* is a chunk of bytes or blocks that hold the redundancy generated from user data (by erasure code, typically XOR operations). The *element* is a unit of data or parity which corresponds to a bit within a code symbol. The *stripe* is a set of data or parity elements that can be referred as a codeword in the coding theory terminology. In addition, a set of elements in a stripe stored in the same disk is called a *strip*, which is known as a code symbol. The disk array system is a collection of a "pile" of multiple stacks that fills the disk's capacity. Note that a stack is a collection of many stripes. A *horizontal code* is an array code in which data and parity elements are in the same stripe but in the separate strip as shown in Fig.1.3. A *vertical code* is an array code in which each strip contains both data and parity within a stripe, or there are both data and parity strips stored in one disk.

1.1.2 Backgrounds of the RAID levels and erasure-correcting codes

RAID or the Reliable/Redundant Arrays of Inexpensive/Independent Disk system is now an industry standard. It is a popular classification for disk arrays [32] which was first introduced as RAID0 in the late 1980s. Since then there are many versions of RAID using techniques based on replication and erasure coding that have been introduced to allow the recovery of disk failures and to provide high reliability. Instead of providing only a single disk, RAIDs employ an array of independent disks, accessed in parallel to collectively achieve a high throughput.

- RAID0: This does not provide redundancy or any fault tolerance, but only improve performance by providing additional storage and maximizing the access speed. The technique used in this RAID0 is solely on striping for load balancing purposes. The probability of disk failures increases when the number of disk drives increases.
- RAID1: Data is written and stored in the redundant disk known as *mirroring disk*. Whenever data is written into one disk, the same data is also written into a redundant one, so that it uses twice as many disks as a non-redundant disk array. This offers the benefit of reliability at the cost of doubling the storage space.
- RAID2: This RAID can tolerate one erasure using a Hamming code. Three parity disks are required to protect four data disks. So, its redundancy is one less than mirroring.
- RAID3: This RAID employs single parity check coding scheme that can recover 1 disk failure. However, in this level data is conceptually interleaved bit-wise over the data disks and a single parity disk is added

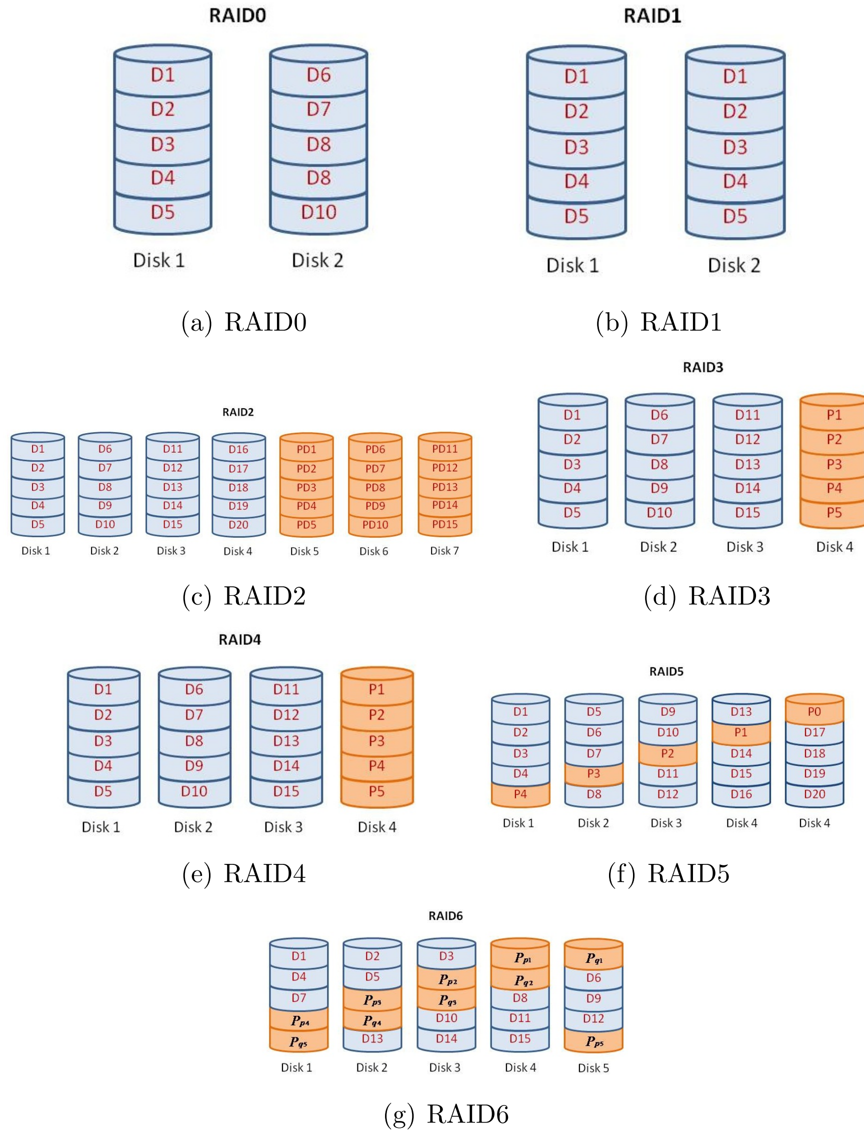
to tolerate any single disk failure. From Fig.1.4(d), the parity disk stores the XORing data from all data disks; for example, $P1 = D1 \oplus D6 \oplus D11$.

- RAID4: This can handle one erasure by using a block-interleaved parity disk array which is similar to the bit-interleaved parity disk array but data is interleaved in blocks rather than in bits. So, in Fig.1.4(e) each data D and parity P is represented in block. The size of these blocks is called the striping unit. Parity is easily computed by XORing the new data for each disk. It is similar to RAID3 in that $P1 = D1 \oplus D6 \oplus D11$, but since it is in a larger size, this parity disk may easily become a bottleneck.

- RAID5: The fault tolerance is covered by the capacity of one disk among N disks, but this level reduces the problem of a bottleneck in RAID4 by using the block-interleaved distributed parity disk array. The advantage of this method is that data are distributed over all of the disks rather than over all but one, so it allows all disks to participate in read/write operations. As shown in Fig.1.4(f), a parity disk $P0$ is computed by XORing data over stripe units $D1, D5, D9$, and $D13$. This property also reduces disk conflicts in the large requests. Even when a single disk fails, data can still be recovered from the parity information that reside in the rest of the disks.

- RAID6: This RAID level provides fault tolerance up to two erasures by providing $P + Q$ redundancy. It is different from RAID5 as it has two additional disks to recover the loss of two disks. This RAID level utilizes several different types of erasure coding techniques such as Reed-Solomon (RS) code, EVENODD code, or X-code. However, each code has its own limitation which will be discussed later.

To summarize, RAID0 solely provides an organization of all stripes on the disks in order to balance load for performance purpose. RAID1 can protect one



(a) RAID0 (b) RAID1

(c) RAID2 (d) RAID3

(e) RAID4 (f) RAID5

(g) RAID6

Figure 1.4: The structure of disk arrays of the standard RAID technology

erasure by using a "mirroring" technique, whereas RAID2, RAID3, and RAID4 can also protect one erasure but using various techniques of coding. RAID6 can tolerate two erasures by providing double parity disks constructed from special designed parity codes such as EVENODD code, or Reed Solomon (RS) code. To achieve a highly available and reliable RAID system, the technique of bitwise parity checking is heavily exploited to correct errors and tolerant disk failures.

RAID performance is evaluated from the update complexity and the number of check disk overheads [33]. The *update complexity* refers to the number of XOR operations required for encoding and decoding if there is at least one disk failure. Additionally, the *encoding/decoding complexity* is also used to measure the complexity of the code construction by monitoring the number of XOR operations the code uses when encoding and decoding.

The Markov chain reliability models are also used to estimate the *mean time to data loss (MTTDL).* To compute the $MTTDL$, two important parameters: (1) the *mean time to failure (MTTF)*, and (2) the *mean time to repair (MTTR)* which is the expected time to recover a system from a failure, are used based on the reliability model [32]. Let the disk failure rate be λ and the repair rate be μ, so that $MTTR = 1/\mu$ and $MTTF = 1/\lambda$. For example, in the single error-correcting RAID, where there are n_G disk-array groups each with G data disks and 1 check disk, we can compute the $MTTDL$ as follows [34]:

$$MTTDL = \frac{(MTTF_{disk})^2}{n_G G(G+1) MTTR_{disk}} \tag{1.1}$$

The strengths and weaknesses of the afore-mentioned standard RAID levels are shown in Table 1.1. There are special RAIDs called *"Nested (Hybrid)*

RAIDs" [18], which provide redundancy by combining two or more of the standard levels of RAID. For example, RAID 0+1 (or RAID 01) is used for both replicating and sharing data among disks by building from many chunks of RAID0 and then mirroring (RAID1), and RAID 1+0 (or RAID 10) provides fault-tolerance by creating a striped set from a series of mirrored drives, but it still has the same cost problem as RAID1. The difference between RAID 01 and RAID 10 is the location of RAID system: RAID 01 is a mirror of stripes while RAID 10 is a stripe of mirrors [18]. Moreover, the RAID parity (or RAID s) provides an error-protection scheme called *"parity"* by simple XOR operations. Although, this special RAIDs may better fault-tolerance than the standard ones, they also increase the complexity for implementation.

1.2 Coding Theory for Data Storages

This section will investigate the practical and popular existing erasure codes in disk storages for tolerating disk erasures/failures. An erasure code is designed to recover the erasures (i.e. bits loss or erased) rather than to correct errors (i.e. bits altered or flipped). The key property of an (n, k) erasure code, which encodes k parts of source data to a total of n parts of encoded data and which guarantees to correct e erasures, is that the original k parts of data can be reconstructed from any $(n-e)$ parts of encoded data. The number of erasures that can be recovered is upper bounded by $e \leq d_{min} - 1$, where d_{min} is the minimum distance of the code.

The optimal (n, k) erasure codes have the property that any k out of n coded bits/data are sufficient to recover the original message. An optimal erasure code is known as a *maximum distance separable* (MDS) code, since its minimum distance is $d_{min} = n-k+1$, the largest possible distance promised by the theory. In this work, we study 3 types of erasure codes that are relevant to

RAID Levels	Strengths	Weaknesses
RAID0	Highest performance	No data protection, any disk fails results in data loss
RAID1	Very high performance and data protection, very minimal penalty on write performance	High redundancy cost overhead, wasteful in storage capacity
RAID2	Previously used for RAM error environments correction (known as a Hamming Code) and in disk drives before the use of embedded error correction	No practical use; Same performance can be achieved by RAID3 at lower cost
RAID3	Excellent performance for large, sequential data requests	Not well-suited for transaction-oriented network applications; Single parity drive does not support multiple, simultaneous read and write requests
RAID4	Data striping supports multiple simultaneous read requests	Write requests suffer from same single parity-drive bottleneck as RAID3 and RAID5 offers equal data protection and better performance at same cost
RAID5	Best cost/performance for transaction-oriented networks; Very high performance, very high data protection; Supports multiple simultaneous reads and writes; Can also be optimized for large, sequential requests	Write performance is slower than RAID0 or RAID1
RAID6	Allows up to two hard drives to crash, high availability solutions	Require a minimum of 5 drives, servers with large capacity requirements

Table 1.1: Strengths and weaknesses of standard RAID levels

data storage networks which are Reed-Solomon (RS) codes, low-density parity check (LDPC) codes, and array codes.

1.2.1 Reed-Solomon (RS) Codes

Reed-Solomon (RS) codes are the most well-known and the most used MDS codes in communications. They achieve the Singleton bound with equality, $d_{min} \leq n - k + 1$, and are therefore MDS. RS codes were first introduced in 1960 by Reed and Solomon. This code construction is based on Galois Field ($GF(2^W)$) operation for W is positive integer. RS codes provide a wide range of code rates from 0 to 1. However, Galois Field arithmetic is rather complex, especially for large fields. RS codes are generally considered not very scalable.

A simplified construction [19] of RS codes for data storage is described in the form of Vandermonde matrices assuming there are m data symbols and e erasures (where $m + e \leq 1 + 2^n$). The length-$(m + e)$ codeword is computed by multiplying the length-m vector of the data by an m-by-$(m + e)$ coding matrix. In addition, this code can be made systematic by simple row reduction of the coding matrix, which diagonalizes the initial m-by-m portion of the matrix. So, the encoding matrix can be constructed by an m-by-m identity matrix followed by an m-by-e checksum computation matrix. Note that a Vandermonde matrix is a type of matrix that has a geometric progression in each row as shown below.

$$\begin{bmatrix} 1 & \alpha_1 & \alpha_1^2 & \cdots & \alpha_1^{n-1} \\ 1 & \alpha_2 & \alpha_2^2 & \cdots & \alpha_2^{n-1} \\ 1 & \alpha_3 & \alpha_3^2 & \cdots & \alpha_3^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \alpha_m & \alpha_m^2 & \cdots & \alpha_m^{n-1} \end{bmatrix}$$

Further, an Vandermonde matrix has the property that any square submatrix has full rank and is invertible.

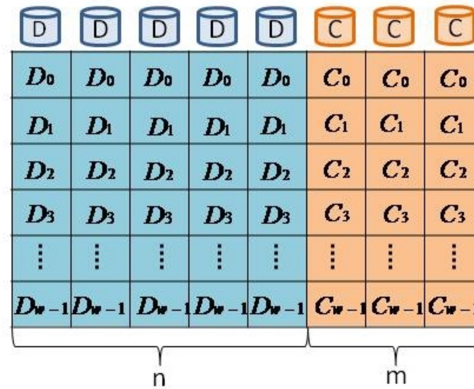


Figure 1.5: The Reed Solomon (RS) codes for disk arrays

For disk array application that targets n data disks and m parity disks such that the entire pool of $(n+m)$ disks can tolerate any m disk failures, the RS code must be defined in $GF(2^W)$ where $2^W \geq n+m$. An illustration is shown in Fig. 1.5.

RS codes in general require complex GF arithmetics. Although, the Vandermonde matrix representation makes encoding and decoding of an RS code a little simpler than otherwise, it nevertheless remains a dense code. Hence, every time fresh data written into the disks or data gets modified, many associated disks need to be read in order to compute the new parity. This causes severe impairments to the computation load and especially the input/output (I/O) throughput of the system.

1.2.2 LDPC Codes

A class of linear block codes called the *low-density parity check (LDPC) code* was first introduced by R. Gallager in the early 1960s [20]. The codes are constructed using bipartite graphs and promise performance closed to the Shannon limit. In a bipartite graph representation, a set of vertices represented columns in an LDPC parity check matrix and another set represent rows. The ith left vertice (variable nodes) is linked to the jth right vertice (check nodes), if and only if there is "1" in the jth row and ith column of the parity check matrix. A good LDPC code usually require a large girth, g, which is the smallest cycle in the graph. A large girth improves the decoding performance of the sum-product algorithm.

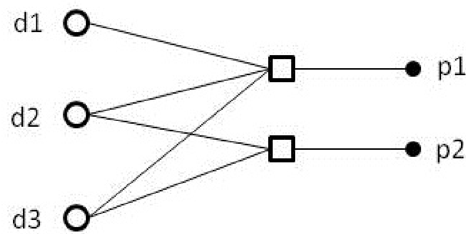


Figure 1.6: An example of a simple LDPC code with $n = 3, m = 2$

LDPC codes can be encoded and decoded by using simple XOR operations. LDPC codes are shown to be asymptotically optimal codes which means that they achieve the Singleton bound when $n \to \infty$. However, for small values of n, such as a few or a few tens, an LDPC code is far from MDS. An example of a bipartite graph describing a simple LDPC code is shown in Fig.1.6. There are $n = 3$ data disks d_1, d_2, and d_3, and $m = 2$ parity disks which are computed by XORing the data disks. The parity p_1 is computed by XORing d_1, d_2, and d_3, while the parity p_2 is computed by XORing d_2 and d_3. The corresponding parity matrix is shown below.

$$H = \left[\begin{array}{ccc|cc} 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 \end{array}\right]$$

This parity check matrix, H, has dimension $m \times (m+n)$ for a $(5,3)$ code. However, for a matrix to be called *low-density*, the number of 1's in the matrix should be sparse. In general, there are two types of LDPC codes that have been described in the academic literature.

1. Regular LDPC Codes: A LDPC code is called (w_c, w_r)-*regular* if a parity matrix H contains exactly w_c 1's per column and $w_r = w_c \frac{n}{m}$ 1's per row, where $w_c \ll m$.

2. Irregular LDPC Codes: If the number of 1's per row or per column is not constant, the code is called an *irregular* code.

Irregular LDPC codes usually outperform regular LDPC codes for very large code lengths.

The code rate of the LDPC code is $R = \frac{n}{n+m}$. The overhead factor (f) is defined as the average number of fn of disks that need to be accessed to reconstruct the n lost data disks (note that $f > 1$). A carefully optimized irregular LDPC code can become space optimal $(f \to 1)$, when the size of n goes to infinite $(n \to \infty)$.

1.2.3 Parity Array Codes

Recently, a class of very promising codes based solely on XOR operations, while maintaining good storage efficiency, are introduced when carefully designed their performances can be optimal or nearly optimal. Thus, these codes are more efficient and ubiquitous than the RS code in terms of computation complexity.

Definition 1.2.1. *An array code* is an erasure-correcting code that is solely computed by simple binary XOR operations. The information and parity(redundant) bits are placed in a two-dimensional array of size $(m \times n)$ rather than a one-dimensional vector.

In an array code, data- and parity-bits are usually represented in a 2-dimensional array. Each column can be viewed as a disk, while each row can be viewed as a strip of the disk. There are 2 types of parity array codes: (1) the horizontal parity array codes where disks store all data or all parity, and (2) the vertical parity array codes where all devices store both data and parity. The vertical parity array codes are more preferable since they have symmetry, such that encoding/decoding complexity is distributed evenly across the disks. An example of an array code with one parity row that can recover from any two column erasures is given below. The first row contains pure information bits and the second row contains parity bits that are computed from the information bits as specified. Hence this code is a vertical parity array code that involves 4 disks altogether with a code rate of $\frac{1}{2}$, and can tolerate 2 concurrent disk failures.

a	b	c	d
$c \oplus d$	$d \oplus a$	$a \oplus b$	$b \oplus c$

For the decoding process, for example, if disk (column) 1 and 3 are lost, recovering data a and c can be done by the following computations.

$$a = d \oplus (d \oplus a) \tag{1.2}$$

$$c = b \oplus (b \oplus c) \tag{1.3}$$

EVENODD Codes

EVENODD codes are known as the "grandfather" of array codes introduced in 1995 [7]. This code is a horizontal MDS array code which can protect and recover 2 erasures. The code word is two-dimensional horizontal and geometrical array with two additional parity columns: one horizontal strip and the other along the diagonals through the stripe. However, the number of data columns (p) needs to be a prime number. The number of rows is $r=p-1$, and the strip count is $n=p+2$. The layout example for $p = 3$ in Table 1.2 shows the basic construction. The code is a $(5,3)$ MDS code defined by a 2×5 array.

$d_{0,0}$	$d_{0,1}$	$d_{0,2}$	P_0	Q_0
$d_{1,0}$	$d_{1,1}$	$d_{1,2}$	P_1	Q_1

Table 1.2: An example of a simple EVENODD code

The first parity is computed by $P_i = \bigoplus_{j=0}^{p-1} d_{i,j}$, where $0 \leq i \leq p-2$. To compute the second parity (Q), we first compute the syndrome (S), which is $S_i = \bigoplus_{j=0}^{p-1} d_{p-1-j,j}$, and then $Q_i = S \oplus \bigoplus_{j=0}^{p-1} d_{i-j,j}$.

X-Codes

The X-code was presented as a nother simple optimal MDS code. This code is a vertical parity array code constructed by $m = 2, n = p - 2$, where p is the prime number. The $(n + 2) \times p$ code array is represented by n rows of data, 2 rows of parity, and $p{=}n + 2$ columns. So X-code can correct 2 erasures [8].

An example of $p = 5$ is given in Table 1.2.3. The left plot shows the pattern of computing the first row of parity elements, while the right plot shows the second row of parity elements. Each parity element is represented by an upper case letter and such a parity element is computed by XORing the set of data elements labeled by the corresponding lower case letter.

(a) The first row of parity elements

a	b	c	d	e
b	c	d	e	a
c	d	e	a	b
D	E	A	B	C
*	*	*	*	*

(b) The second row of parity elements

a	b	c	d	e
e	a	b	c	d
d	e	a	b	c
*	*	*	*	*
C	D	E	A	B

Table 1.3: An (5×5) array of X-code

From the construction of X-code, it is clear that two parity rows are independently obtained, and each information bit affects only one parity bit in each parity row. Thus, all parity bits depend solely on information bits, but not from among themselves. The update complexity is exactly 2 since a single data bit needs only updating in two parity bits [8].

The Row-Diagonal Parity (RDP) Codes

The row-diagonal parity (RDP) code [37] is proposed as a double fault tolerant array code, which like the X-code, is also a variation of the EVENODD code. The code is described by a $(p-1) \times (p+1)$ array, where p is a prime number greater than 2. It provides the last two columns as two parity columns, so the first $p-1$ columns contain information bits.

(a) The first column of parity elements

a	a	a	a	A	*
b	b	b	b	B	*
c	c	c	c	C	*
d	d	d	d	D	*

(b) The second columns of parity elements

d	c	b	a	*	D
c	b	a	*	d	C
b	a	*	d	c	B
a	*	d	c	b	A

Table 1.4: An (4×6) array of RDP code

Table 1.4 illustrates an example layout to construct two columns of parity of (4×6) RDP code. The first and second parity columns are named as the row parity column and the diagonal parity column, respectively. Also, each parity element is represented by an upper case letter and such a parity element is computed by XORing the set of data elements labeled by the corresponding lower case letter. In this code, the missing diagonal does not have a corresponding diagonal parity.

Array codes that can correct more than 2 erasures

For large systems, array codes which can handle more than two erasures are required to improve the reliability of disk storage. Here are some examples that are MDS or nearly MDS codes.

1. STAR codes: This MDS code is an extended version of EVENODD codes that protects 3 erasures [35].

a	*	d	c	b	*	*	A
b	a	*	d	c	*	*	B
c	b	a	*	d	*	*	C
d	c	b	a	*	*	*	D

Table 1.5: An (4×8) array of STAR code

The first two parity columns are computed the same as the ones of EVENODD codes by using the syndrome, so without the third parity column the STAR codes are just the EVENODD codes. The third parity is computed by XORing the information symbols within the diagonal line of slope -1 as shown in Table. 1.5.

2. WEAVER codes: This code is a vertical parity array code that can tolerate higher failures. There exist specific realizations of WEAVER codes of $m = 2, n = 2$ and $m = 3, n = 3$, which tolerate double and triple disk failures, respectively, and are MDS. However, WEAVER codes in general are not MDS codes.

3. HoVer codes: This code is a combination of horizontal and vertical parity array code. The general parameters of HoVer codes are $HoVer^{t}_{v,h}[r, c]$, where t is the number of fault tolerance this code can handle, v is the number of coding rows (vertical parity), h the number of coding columns (horizontal parity), r is the number of data rows, and c is the number of data columns. All parameters are illustrated with the array structure in Fig. 1.7. Even through, this code is not an MDS code, it is still interesting as it provides good flexibility in code design. This code is also known as turbo product code or block turbo code [9].

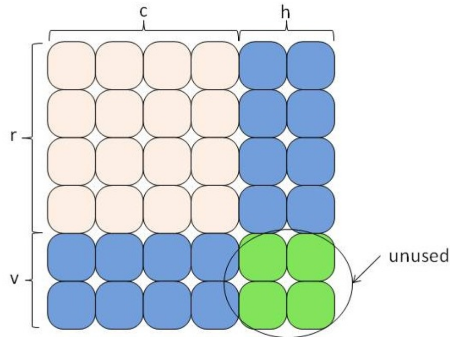


Figure 1.7: The $HoVer^{t}_{v,h}[r, c]$ codes.

4. B-Codes: A novel technique to construct an MDS array code using a perfect 1-factorization (P1F) of the graph theory is introduced and proposed in [8]. This code has dimension $n \times 2n$, where n is an integer greater than 2. The first $n - 1$ rows store information bits, while the bits in the last row are parity bits. Because of the property of P1F technique, any two information bits are not used to compute any pair of parity bits and result in each information bit is protected by exactly 2 parity bits contained in other columns. This code reaches the optimal update complexity.

   B-codes achieve the Singleton bound, so they are optimal in terms of space efficiency. There are a lot of researches inspired by this code and they are presented in [15], [53], [40], to name but a few.

Table 1.6 summarizes these three types of important erasure codes: RS codes, LDPC codes, and array codes.

Erasure code	Characteristics
Reed-Solomon Codes	1. Optimal MDS code, space efficient 2. Flexible code length and rate since it works for any n and m 3. Use GF (Galois field) operations, computationally complex, and hence expensive 4. Dense code, so I/O throughput can be poor
LDPC Codes	1. Binary encoding/decoding 2. Good performance at long code lengths 3. Less structural (hardware implementation can be tricky) 4. Performance is far from optimal at short lengths (storage systems use short erasure codes)
Array Codes	1. Well structured 2. Space efficient 3. Binary encoding and decoding (suitable for hardware implementation) 4. There are not many MDS array codes, and most of them correct only 2 to 3 erasures

Table 1.6: Erasure codes for disk storage arrays

1.3 Flash Drives

The past decade has witnessed an explosive growth in semiconductor memories, especially the flash memory, driven by cellular phones and other electronic portable devices such as GPS and MP3 players. The semiconductor memories are divided into two branches which are based on the complementary metal-oxide-semiconductor (CMOS) technology as shown in Fig. 1.8.

This section will explain the technology of flash memories, an important class of solid-state memory, and their current trend in industry fields. Flash

memory is a particular type of EEPROM or Electronically Erasable Programmable Read Only Memory. It is a non-volatile memory that maintains stored information without requiring a power source. Compared to the hard disks and optical disks which provide two distinctive states to represent 0s and 1s, flash memories have many levels of cell states that can represent the digital data. To increase the cell state level can be achieved by injecting the electron into the cell level is easy, but to decrease its level is both costly and slow since it has to erase the whole block. Furthermore, frequent block erasing can deteriorate the life time of flash memories since the overall life time is limited by the counting of erase operations.

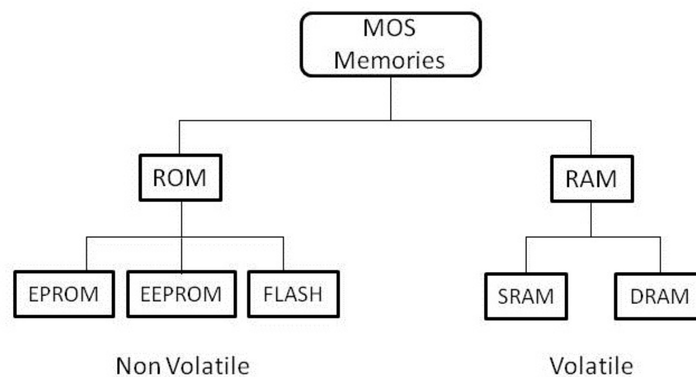


Figure 1.8: MOS memory tree

There are 2 different types of flash memories in terms of logical technologies to map data: NAND and NOR flash memories which are described in the following.

- NOR Flash: In NOR flash memory, a standard MOSFET is resembled in each cell and each cell has two gates which are stacked vertically. The common drain connection called bit line is connected to each cell and can be read directly in order to fast read for the fast program execution.

- NAND Flash: The memory cells are connected in series and also connected to the bit line and source line through two selected transistors in order to increase capacity and decrease cost of flash memory. It has a smaller cell size and lower die cost than NOR flash [44].

In both types, write operations can only clear bit or change their cell value from 1 to 0. To set bit or change their cell value from 0 to 1 needs to erase an entire block of memory [51]. Since each bit in a NOR flash is cleared once per erase cycle, it suffers from high erase times. Unlike a NOR flash, a NAND flash is not directly addressable by the processor. It is accessed by a page (or block). However, after a page is full, an erase cycle must be required. Because of this properties, the storage management techniques for each type of flash memory are different from the magnetic disks.

1.3.1 NOR vs. NAND Flash Memory

There are some difference between NOR and NAND flash memories because of their performance and different using propose. NOR flash is very similar to a Random Access Memory (RAM) device and has enough address pins to map its entire media which allows for easy access to each and everyone of its bytes. NAND flash has more complicated I/O interface since it is interfaced to each other serially between bit line that may vary from one device to another or from vendor to vendor. The basic cell in NAND flash is a MOSFET transistor with a floating gate which tunnels a charge during write operations and removes during erasing operations. NAND Flash, which was designed with a very small cell size to enable a low cost-per-bit of stored data, has been used primarily as a high-density data storage medium for consumer devices such as digital still cameras and USB solid-state disk drives.

NOR flash is suitable and ideal for low-density, high-speed read applications, which are mostly read only, so it is often referred to as code-storage applications. Because code can be directly executed in place, NOR is ideal for storing firmware, boot code, operating systems, and other data that changes infrequently. On the other hand, NAND flash is developed for higher-density data storage, and achieves a smaller cell size that leads to a smaller chip size and lower cost-per-bit since it can connect eight memory transistors in a series. Thus, NAND flash systems perform faster write and erase operations by programming blocks of data, so that it is ideal for low-cost, high-speed program/erase applications and usually referred to as data-storage applications [31].

However, to increase the performance and reliability of flash hardware, the well-designed software strategies are effectively applied. The proposes of flash memory management software include [51]:

1. Avoiding data loss: The most important goal in managing flash memory is to assure that no data is lost due to an interrupted operation or the failure of device. Several techniques can achieve this goal, for example, (i) rewrite operations: new data can be written and verified before the old data is deleted, so that neither power loss nor other interruption can result in the loss of both old and new data, and (ii) bad block management: this is the software management that can prevent data being written to failed memory blocks since it can check which blocks are bad and avoid writing to those block from the beginning. Moreover, at the nearly the end of flash memory life, the good software management can implement a fruitful strategy such as placing the entire flash unit in a read-only state, thereby avoiding data loss when the number of block errors exceeds a predefined number [51].
2. Improving the effective performance: There are two ways to improve the

performance which are compaction and multi-threading. Compaction identifies which block is obsolete or full that can be erased, then copies any valid data to a new location before erasing the blocks to make them available for reuse. Multi-treading system helps to organize read operations by allowing high-priority read requests to interrupt low-priority maintenance operations. It can reduce read latency by orders of magnitude compared to a single-thread solution.

3. Maximizing flash memory life span: The Wear-levelling algorithm is a famous technique that can prevent overuse of memory blocks. It can monitor block usage to identify high-use areas and low-use area containing static data, then swap the static data into the high-use areas. Also, it balances write operations across all available blocks by choosing the optimal location for each write operation.

The decision between NAND and NOR memory will ultimately depend on both technical and pricing requirements of the device being built. Whatever type or combination of flash is used, it is prudent to include memory management software to prevent data loss while improving the performance and maximizing the lifespan of the memory [2].

We can conclude the properties and performances of both NOR and NAND flash memories in Table 1.7.

In data storage applications, NAND flash memory is often used because of its characteristics we described above. However, the major drawback/limitation of NAND flash memory is that it has the limitation in updating (writing) times so it degrades the lifespan of flash memory. In this work, we will apply coding techniques to solve this problem. The objective is to maximize the number of writes before a memory needs to be erased and reset the whole block to be ready to write a new data again.

NAND Flash	NOR Flash
core cells connected in series (normal 8 or 16 cells)	core cells connected in parallel (common ground)
high density	lower density
medium read speed	high read speed
high write speed	slow write speed
high erase speed	slow erase speed
an indirect or I/O like access (good for data storage)	a random access interface (good for code execution)

Table 1.7: The properties and performances of NOR and NAND flash memories

1.4 Outline

In the rest of this dissertation, we will discuss coding techniques for disk arrays in Chapter 2 and for distributed large-scale data centers in Chapter 3, and for flash memory are in Chapter 4.

In Chapter 2, we propose a new class of optimal MDS codes constructed from graphs which can achieve the Singleton bound and which are based only on simple XOR operations. These codes termed complete-graph-of-ring (CGR) codes can recover the maximal disk failure with minimal spare disks and are particularly useful for disk arrays. Additionally, these codes can be considered as a modification of LDPC codes.

Extending the MDS coding results developed in Chapter 2 as well as those proposed in the literature, next in Chapter 3, we tackle the data protection and disk recovery issue in the context of large data centers. Accounting for the possibility of splitting a data center and merging two data centers, we propose layered protection, and develop a nested coding architecture with hierarchical protection for distributed storage networks.

Chapter 4 presents and discusses coding schemes for flash memories. The goal is to improve their life cycles and maximize the number of writing times before a block erasure. Both word-efficient bit-efficient (WEBE) codes and flash marker (FM) codes are introduced and analyzed in terms of the number of bit-writes and the number of word-writes they can guarantee before the block erasure is needed. We present the new code design idea, discuss its feasibility and efficiency, and estimate its performance.

Chapter 5 concludes this dissertation and discusses the future industrial trends of both disk storages and flash memory. In the research work, many coding techniques have been studied, improved, and generated in the pipeline. We can extend our work and develop our codes for various applications.

Chapter 2

MDS codes for disk arrays

This chapter presents practical coding techniques for data disks in order to combat disk failures or erasures. We investigate various types of array codes, due to their simplicity and high I/O throughput. Since maximum distance separable codes are space optimal, we focus on graph constructions of MDS array codes or nearly-MDS array codes. We also study MDS array codes in the form of "low-density (sparse)" matrices and propose the algorithm for encoding/decoding.

2.1 Introduction

Storage of digital data has become a necessary part of our life in today's information age. Huge volumes of data information are created, transferred, and stored everyday. Reliable and fast-recovery data storage is essential to support data availability, persistence, and integrity. Various techniques for increasing storage reliability have been actively exploited, including powerful

error correction codes applied inside each block/sector of a disk to protect against bit errors or bit loss, and, more recently, efficient erasure codes applied between disks (or blocks and sectors) to protect against a disk (block/sector) failure [3]-[13]. The latter, generally referred to as redundant/reliable arrays of inexpensive/independent disks, or, RAID, is becoming an important industrial standard [16]. A key technical challenge of RAID is the design of efficient erasure codes that can recover a target number of device failures with minimal redundancy, namely, maximum distance separable codes. An array code is not always MDS, but an MDS array code is particularly desirable for combating data loss caused by disk failure in disk arrays. The properties of MDS codes will be discussed later in the next section.

2.1.1 MDS Codes and Their Properties

Space-optimal or MDS codes have several desirable properties.

Theorem 2.1.1. [35] Consider an (n, k) error correcting code that encodes k message (data) symbols to n codeword symbols, where $n \geq k$. Such a code can usually tolerate a loss of e symbols during transmission, where $e \leq n - k$. When $e = n - k$, the code meets the Singleton Bound, and is called an MDS code.

Theorem 2.1.2. [36] An (n, k, d) code C with a generator matrix $G = [I, A]$, where I is a rank-k identity matrix and A is a $k \times (n - k)$-matrix, is an MDS code if and only if every square sub matrix of A is nonsingular.

Theorem 2.1.3. Let C be an (n, k) linear code with minimum distance d_{min}, then the following statements are equivalent:

1. C is an MDS code.
2. The code C' dual to C is an MDS code.
3. $d_{min} = n - k + 1$.
4. The code can correct any set of $e = n - k$ erasures.
5. Any k columns of the k-by-$(n - k)$ generator matrix for C are linearly independent.
6. If a generator matrix for C is in the standard form $[I, A]$, then every square submatrix of A is nonsingular.
7. Given any d_{min} coordinate positions, there is a (minimum weight) code word whose non-zero entries are in precisely these positions.

The MDS codes achieve the largest possible minimum distance (d_{min}) among linear codes of the same size, and therefore provide the best data loss recovery capability within a given code size.

2.1.2 Literature Reviews

We provide a quick review of the existing array codes. More detailed discussion can be found in Chapter 1.

The EVENODD codes [7] are the first and the most well-known class of MDS array codes that have inspired many subsequent designs of good array codes. The perspective of this code is to overcome the drawback of traditional array codes which is the linear increase of the update complexity as the number of columns increases. EVENODD codes and their generalizations are designed

based on independent parity columns resulting in a more efficient information update.

However, EVENODD codes have only two logic parity symbols, which means that they can recover up to two disk failures, while the generalized EVENODD code can tolerate three disk failures. Hence, the general question arises as whether it is possible to develop MDS array codes with larger erasure correcting capability and with similar low complexity. X-Codes developed in [8] provides a good answer. X-Codes boast a simple geometrical structure and an update complexity of exactly 2. Since the distance of X-codes is 3, it can recover up to 2 disk failures with lower complexity. Another class of MDS codes, named B-Codes, and their dual codes [3] also have distance 3 and their update complexity is also optimal, which is exactly 2. Additionally, a perfect one-factorization (P1F) of complete graphs is a technique to construct B codes. Both P1F technique and a graphical structure make B-code simply to implement and easy to construct an array code. Efficient decoding algorithms are introduced for both erasure and error correcting for B-codes.

There are some array codes that can tolerate more than 3 erasures. These codes [7, 6, 8, 3] have parity in either horizontal or vertical positions. J. Hafner also introduced the code which has parity bits in both horizontal and vertical positions, termed HoVer codes [9]. The HoVer codes proposed by Hafner can tolerate more than 3 erasures, but unfortunately they are only approximately MDS codes.

This work is inspired by the beauty of graph representation of these array codes, especially B-codes, and their MDS properties. We are interested in finding answers to the following questions. What kind of graphs would lead us to MDS codes? What would be the conditions and the properties of such graphs? How could we construct an MDS code from such a graph? The experiments in [9] suggest that some code settings are MDS codes, but some

are not, while [3] successfully constructs the code based on a specific setting of graphs. Here, we explore the possibility of finding generalized ideas for the array codes based on graph structures as well as looking for systematic ways to construct such codes. Our study results in a new class of MDS codes from a new class of graphs in the next section.

2.2 CGR Codes

In this section, we propose a new erasure code construction technique to reduce disk failures and increase the capability of fault tolerance. The proposed method systematically builds MDS codes from an efficient class of nested graphs, termed *complete-graph-of-rings (CGR)*. The resultant codes, termed "*CGR codes*", and theirs dual codes require minimal encoding/decoding complexity.

2.2.1 Code Construction and Algorithms

The proposed CGR codes are constructed in three steps:

1. Building the appropriate CGR graph of the appropriate parameters
2. Mapping the CGR graph to an array code
3. Reordering by left-cyclically shifting rows (following the perfect 1-factorization technique) in the array code to achieve MDS

The notations and definitions:

Let K_v be a complete graph (or base graph) with v vertices and $\frac{v(v-1)}{2}$ edges, and each vertex has a degree of $v-1$. The ring graph is denoted by C_n with n edges. So, we can define the CGR graph by $CGR(K_{v_1}, C_{v_2})$, where we replace each vertex of a complete graph K_{v_1} with a ring graph C_{v_2}, and replace each edge connecting two vertices in K_{v_1} with a group of v_2 parallel edges connecting the respective vertices in two rings. The examples of $CGR(K_2, C_5)$ and $CGR(K_4, C_7)$ are illustrated in Fig. 2.1.

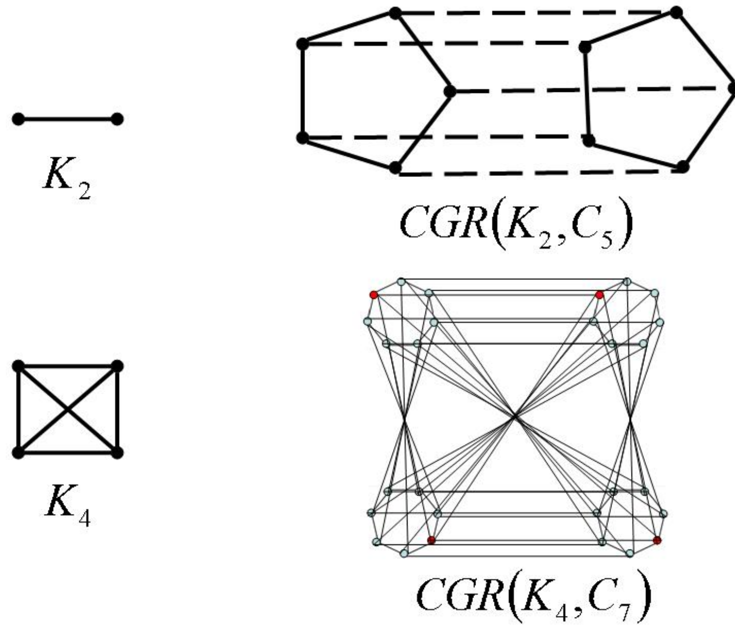


Figure 2.1: CGR graphs constructed from base graphs. Left: base graphs K_2 and K_4; right: resultant CGR graphs $CGR(K_2, C_5)$ and $CGR(K_4, C_7)$.

The sufficient conditions that allow a CGR graph to convert to an MDS code is given as follows.

Theorem 2.2.1. If a CGR graph Υ_{v_1,v_2} constructed from a complete graph K_{v_1} and a ring graph C_{v_2} satisfying the following conditions: (1) v_1 is even, and (2) $v_2 = v_1 + 3$, then there exists a way to place all the vertices and

edges in an array of $\frac{v_2 v_1}{2} \times v_2$. When the vertices are interpreted as data bits and the edges connecting two vertices are interpreted as parities associated with two data bits, the resultant array defines an array code of parameters $(N, K, d_{min}) = (v_1, 2, d_{v_1} - 1)$ capable of correcting up to $(v_2 - 2)$ erasures. Its dual code is a $(v_2, v_2 - 2, 3)$ MDS code capable of correcting up to 2 erasures.

We now present the detailed algorithms for each of the three steps in constructing an MDS CGR code.

Algorithm 1: Graph Construction and Labeling

This algorithm constructs a (v_1+1)-regular CGR graph Υ_{v_1,v_2} from a complete graph K_{v_1} and a set of v_1 rings C_{v_2}, where v_2 is even and $v_2 = v_1 + 3$.

1. Take a set of v_1 number of rings C_{v_2}. Label the vertices of the first ring counter-clockwise as $0, 1, \cdots, v_2 - 1$; label the vertices of the next ring similarly as $v_2, v_2 + 1, \cdots, 2v_2 - 1$, and so on, until all the rings are labelled. We have altogether v_1 rings or v_1 sets of vertices, where the vertices of the jth ring are labelled by $\mathbf{V}_j = \{jv_2, jv_2{+}1, \cdots, (j{+}1)v_2{-}1\}$, for $j = 0, 1, ..., v_1 - 1$.
2. Each edge inside a ring, termed a *ring* edge, is marked by the pair of vertices on both ends. We have altogether v_1 sets of ring edges, where the edges of the jth ring are labelled by $\mathbf{E}_j = \{(jv_2, jv_2{+}1), (jv_2{+}1, jv_2{+}2), ..., ((j{+}1)v_2{-}2, (j{+}1)v_2{-}1), ((j{+}1)v_2{-}1, jv_2)\}$, for $j = 0, 1, ... v_1 - 1$.
3. For any pair of rings, connect their indexes using v_2 parallel *inter-ring* edges, such that the lowest index of one ring is connected to the lowest index of the other, the next lowest is connected to the next lowest, and

so on. We have altogether $v_1(v_1-1)/2$ sets of inter-ring edges, labelled respectively as $\mathbf{E}_{i.j} = \{(iv_2, jv_2), (iv_2+1, jv_2+1), ..., ((i+1)v_2-1, (j+1)v_2-1)\}$, for $0 \leq i < j \leq v_1 - 1$.

Example 1: An example of labeling the vertices for CGR(K_2,C_5) is shown in Fig. 2.2. Each vertex has 2 ring edges and 1 inter-ring edges connecting between rings. This graph possesses many desirable properties, including symmetry and regularity (all vertices have the same number of degree 3).

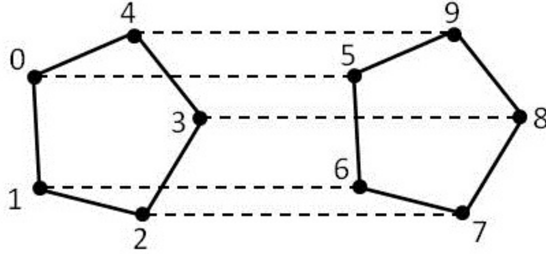


Figure 2.2: Labeling of 3-regular CGR(K_2,C_5).

Algorithm 2: CGR array code construction

This process describes how to map CGR graph codes constructed by Algorithm 1 to arrays. We map the vertices to information bits and edges to parity bits, which can be computed by XORing two information bits on both ends of the edge. Let us consider constructing a CGR array code using CGR(K_{v_1}, C_{v_2}) labelled by Algorithm 1. The array code will consist of $v_1(v_1+3)/2 = v_1v_2/2$ rows and v_2 columns (recall $v_2 = v_1 + 3$ and v_1 is an even integer).

1. The v_1 sets of vertices, each corresponding to a ring, are placed in the

first v_1 rows as systematic bits. By default, the vertices in each set is placed in ascending order from left to right to form a row.

2. The v_1 sets of ring edges, each corresponding to a ring, are placed in the next v_1 rows as parity bits. By default, the edges of the same ring are placed in ascending order, with the one connecting the two smallest indexes being the first, and the wrap-around edge that connects the biggest index and the smallest index being the last.

3. The $v_1(v_1-1)/2$ sets of inter-ring edges, each connecting a pair of rings, are placed in the remainder $v_1(v_1-1)/2$ rows as parity bits. The edges in each set is placed in ascending order, with the one connecting the two smallest indexes being the first, and the one connecting the largest indexes being the last.

4. Next, cyclically shift the elements in each row according to an *offset vector*. An offset vector is a pre-determined vector in the form of

$$(\alpha_0, \alpha_1, ..., \alpha_{(v_1 v_2)/2-1}) \in \{0, 1, ..., v_2 - 1\}^{(v_1 v_2)/2}.$$

   Cyclically shift the jth row *to the left* by α_j positions, or, equivalent, strip off the first α_j elements in the jth row and append them to the end of row. When the offset vector is appropriately designed, such as using Algorithm 3, then the array code is MDS.

Example 2: Consider CGR(K_2, C_5) with vertices labelled from 0 to 9 as shown in Fig. 2.2. According to Algorithm 2, we can place all the vertices (information bits) and edges (parity of two information bits) in a (5×5) array, with the first 2 rows for vertices, the next 2 rows for ring edges and the last one row for inter-ring edges. Suppose that we are given an offset vector $(0, 1, 2, 2, 4)$, then these five rows should be cyclically shifted by 0,1,2,2,4 positions to the left, respectively, giving rise to the following arrays:

Before cyclic shifting:

0	1	2	3	4
5	6	7	8	9
$\mathbf{0} \oplus \mathbf{1}$	$2 \oplus 3$	$3 \oplus 4$	$4 \oplus 0$	$1 \oplus 2$
$\mathbf{5} \oplus \mathbf{6}$	$7 \oplus 8$	$8 \oplus 9$	$9 \oplus 5$	$6 \oplus 7$
$\mathbf{0} \oplus \mathbf{5}$	$4 \oplus 9$	$1 \oplus 6$	$2 \oplus 7$	$3 \oplus 8$

After cyclic shifting:

0	1	2	3	4
6	7	8	9	**5**
$2 \oplus 3$	$3 \oplus 4$	$4 \oplus 0$	$\mathbf{0} \oplus \mathbf{1}$	$1 \oplus 2$
$7 \oplus 8$	$8 \oplus 9$	$9 \oplus 5$	$\mathbf{5} \oplus \mathbf{6}$	$6 \oplus 7$
$4 \oplus 9$	$\mathbf{0} \oplus \mathbf{5}$	$1 \oplus 6$	$2 \oplus 7$	$3 \oplus 8$

Algorithm 3: Offset Vector Determination

This algorithm determines the offsets for the rows of the inter-ring edges of CGR(K_{v_1}, C_{v_2}), by applying P1F on a larger complete graph K_{v_1+2}, and then trimming it down to K_{v_1}.

1. First label the vertices in K_{v_1+2} with $0, 1, ..., v_1 - 1$ and $-\infty$ and $+\infty$, where v_1 is even.
2. Place all the vertices in a wheel, with $-\infty$ in the center, and all the others in a ring (spaced evenly) surrounding the center. Connect any pair of vertices with an edge.

3. Apply the well-known P1F technique discussed in [12],[53] to group all the edges of K_{v_1+2} in v_1+1 factors, such that each factor consists of a center-pointing edge (i.e. edge $(-\infty, i)$ where $i \in \{0, 1, ..., v_1 - 1, \infty\}$) and a set of $v_1/2$ edges that are diagonal to (" perpendicular to") it.

4. Assign the $(-\infty, \infty)$ group an offset v_1+2, and assign to the other groups distinct offsets chosen arbitrarily from $0, 1, ..., v_1 - 1$.

5. Remove from each factor the edges that are incident with vertices $-\infty$ or ∞. What remains are all the edges from the base graph K_{v_1} and their corresponding offsets, which are the offsets for all the inter-ring-edge-rows.

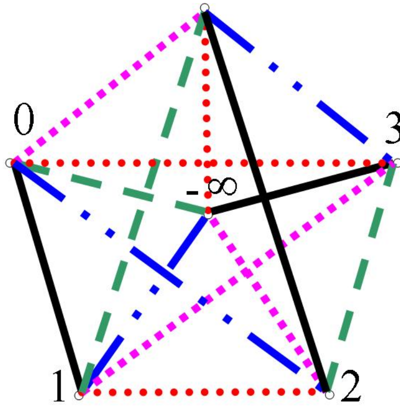


Figure 2.3: Complete graph K_6.

Example 3: Consider CGR(K_4, C_7). To determine the offsets for the inter-ring-edge-rows, consider P1F on K_6, as illustrated in Fig. 2.3. The vertex $-\infty$ is placed in the center, and the vertices $0, 1, 2, 3, +\infty$ may take arbitrary positions in the cycle. The P1F partitions all the edges in 5 factors as shown below.

center-point edge	diagonal edges	offsetA	offsetB
$(-\infty, +\infty)$	$(0,3)$, $(1,2)$	6	6
$(-\infty, 0)$	$(1,+\infty)$, $(2,3)$	1	0
$(-\infty, 1)$	$(0,2)$, $(3,+\infty)$	3	1
$(-\infty, 2)$	$(1,3)$, $(0,+\infty)$	0	2
$(-\infty, 3)$	$(0,1)$, $(2,+\infty)$	2	3

Note that the offset vector for any CGR(K_{v_1}, C_{v_2}) code is not unique, but all of them can generate MDS.

2.3 Proofs of CGR Array Codes

This section shows all proofs of MDS properties of CGR array codes. In addition, the perfect one-factorization technique used to construct the offset vector presents the relation of the inter-ring edges and the flexibility to choose various offset vectors for any CGR(K_{v_1}, C_{v_2}) code.

2.3.1 Proofs of an MDS Property of CGR Codes

Lemma 2.3.1. For any 2 columns of an vertical $(n, 2, n-2+1)$ MDS array code, it is sufficient to recover from the erasures.

Proof. According to the definition and MDS code theorems: A bridged code is a pair of MDS codes with the same structure, let $S1_i$, $P1_i$ be the systematic bits and parity bits of a column in $B1$ respectively, as well as $S2_j$,$P2_j$ be the systematic bits of a column in $B2$. The $P3_{ij}$ is $S1 \oplus S2$. □

Lemma 2.3.2. Given any two columns of a bridged code, one from $B1$ denoted as $S1_i$, another from $B2$ as $S2_j$, and $P3$, it is sufficient to recover from erasures.

Proof. Since $P3 = S2 \oplus S1$, We can obtain $S1_j = S2_j \oplus P3$ and $P1_j = P2_j \oplus P3$. Then, from Lemma 2.3.1, $S1_i$ and $S2_j$ is sufficient to decode. □

Lemma 2.3.3. For a CGR(K_{v_1}, C_{v_2} code, it can be decomposed into v_2 sub codes, each of which is a B-code, which is also an MDS code.

Proof. Apply the CGR-B code shortening algorithm to every ith vertices in the rings, $i = 0, 1, .., v_2 - 1$. Full details are described in Chapter 2, section 2.5. □

Lemma 2.3.4. Given an ith sub B-code, the column of this code resides only in $(i - k) \mod v_1, k = 0, 1, .., v_2 - 1$. And does not occupy 2 consecutive columns in the CGR code.

Proof. The structure of the CGR code is defined by K_{v_1} and C_{v_2}. Since $v_2 = v_1 + 2$, the sub complete graph spans (occupies) only the v_1 column of the CGR code. □

Lemma 2.3.5. Given a column in CGR code, there are 2 consecutive sub codes that do not occupy this column.

Proof. According to Lemma 2.3.4, and from the cyclically shift pattern, for any 2 consecutive sub codes B_i,B_{i+1}, if B_i does not occupy the jth and $(j+1)$th columns, then B_{i+1} does not occupy $(j+1)$th and $(j+2)$th columns. Thus the $(j+1)$th column does not contain any part of the code from B_i and B_{i+1}. □

Lemma 2.3.6. Given 2 columns in the CGR code, there are two possible cases that:

- if the 2 columns are not consecutive, 4 out of v_2 sub codes are not self-sufficiently decodable.
- if the 2 columns are consecutive, 3 consecutive sub codes out of v_2 sub codes are not self-sufficiently decodable.

Proof. It is easy to verify from Lemma 2.3.5 for the first case. Then, for the second case, if B_i, B_{i+1} and B_{i+2} are consecutive sub codes, and jth and $(j+1)$th are the non-occupying columns of B_i, then $(j+1)$th, $(j+2)$th and $(j+2)$th, $(j+3)$th are the non-occupying columns of B_{i+1} and B_{i+2} respectively. Consider $(j+1)$th and $(j+2)$th columns, which are 2 consecutive columns, there are 3 consecutive sub codes B_i, B_{i+1} and B_{i+2} that do not belong to these columns of the CGR code. □

Lemma 2.3.7. It is a necessary condition for a vertical $(n, 2, n-2+1)$ MDS array code that n–1 versions of information bits occupy n–1 out of n columns.

Proof. This guarantees that any two columns will contain at least a version of an information bit, otherwise this information bit will be wiped out an impossible to decode. □

Lemma 2.3.8. For any information bit of a sub B-code, two of the XORed versions of this bit reside in the column where this sub code does not occupy.

Proof. According to Lemma 2.3.7, this fact must hold for both sub code itself and the CGR code. So for the total $v_2 - 1$ versions of an information bit,

$v_1 - 1 = v_2 - 2$ versions must occupy inside the sub code and we have 2 versions left which are fit with the other two remaining columns of the CGR code. □

Lemma 2.3.9. For any 2 columns of the CGR code, there exists a pair or two of consecutive sub codes. The pair forms a bridged code, and thus decodable.

Proof. Since for each column of CGR codes, there are connections that is connected by edges. Thus, if we provide any pair column of a consecutive sub code, there is always a bridge between them and yield to decode. □

Lemma 2.3.10. Any 2 columns of the CGR code is enough for recovering data from erasures.

Proof. We show that all sub codes fall in 2 cases which are decodable.

- the sub code which are self-sufficient, it is decodable.
- the sub code that are not self-sufficient, then decodable by Lemma 2.3.9.

□

Theorem 2.3.11. The CGR code is an MDS code.

Proof. Because 2 columns of the CGR code is sufficiently decodable. Thus, from Lemma 2.3.1, this is an MDS code. □

Example A.1: Let consider a CGR(K_4, C_7) as a ring of complete graph which is simply to prove that this code is achieve an MDS code property. First of all, we rewrite the structure of CGR graph to be in the ring of complete graph (RCG) which makes all inner-edges in CGR be the inter-edges (dotted line) of each complete graph as shown in Fig. 2.4.

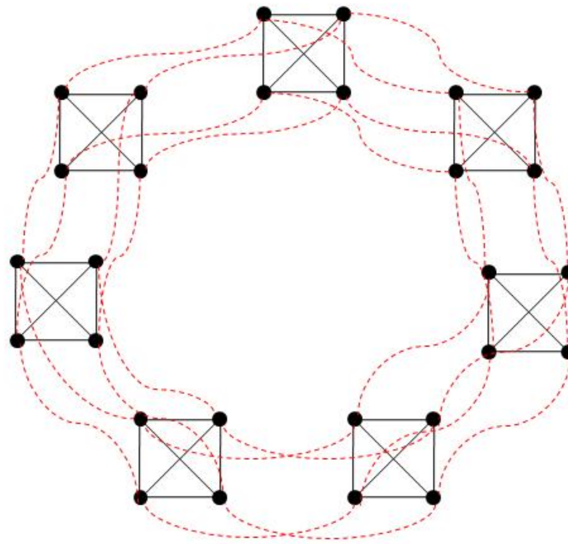


Figure 2.4: A ring of complete graph of (K_4, C_7)

Then, from the Fig. 2.4 the dotted lines show all edges which connect the nodes inside each ring. However, they are closely considered here to prove that CGR codes achieve the singleton bound.

Recall that the array MDS code of CGR(K_4, C_7) code is:

0	1	2	3	4	5	6
8	9	10	11	12	13	7
16	17	18	19	20	14	15
24	25	26	27	21	22	23
4,5	5+6	6+0	0+1	1+2	2+3	3+4
11+12	12+13	13+7	7+8	8+9	9+10	10+11
18+19	19+20	20+14	14+15	15+16	16+17	17+18
25+26	26+27	27+21	21+22	22+23	23+24	24+25
2+9	3+10	4+11	5+12	6+13	0+7	1+8
3+17	4+18	5+19	6+20	0+14	1+15	2+16
6+27	0+21	1+22	2+23	3+24	4+25	5+26
13+20	7+14	8+15	9+16	10+17	11+18	12+19
7+21	8+22	9+23	10+24	11+25	12+26	13+27
15+22	16+23	17+24	18+25	19+26	20+27	14+21

Note that the second set of this array code contains parity bits which are come from XORing between inner-ring edges. In order to understand it more clearly, we can separate and consider this code in terms of B-codes without showing inner-edges. Now, we have $B_0 - B_6$ as shown in Table 2.1- Table 2.7.

From all sub arrays B_0–B_6, we can separately consider them into two cases:

1. *a complete case,* which has either one information bit and three parity bits, or two information bits and two parity bits so that they can completely recover others in the same group of complete graph without asking for any help from inner-ring edges; i.e., column 1 and 2 of B_0, B_1, B_2, and B_3.
2. *an incomplete case,* which has either one information bit and one parity

Table 2.1: B_0

0	-	-	-	-	-	-
-	-	-	-	-	-	7
-	-	-	-	-	14	-
-	-	-	-	21	-	-
-	-	-	-	-	-	-
-	-	-	-	-	-	-
-	-	-	-	-	-	-
-	-	-	-	-	-	-
-	-	-	-	-	0+7	-
-	-	-	-	0+14	-	-
-	0+21	-	-	-	-	-
-	7+14	-	-	-	-	-
7+21	-	-	-	-	-	-
-	-	-	-	-	-	14+21

bit, or no any bit survived; i.e., column 1 and 2 of B_4, B_5, and B_6, so that this complete graph cannot recover itself. Now, we will only consider the second case which is the worst case where we need some help form the inner-ring edges.

We can notice that for any set of B if we consider the worst case of this array which there are only two columns left (2 survivors) for recovering all information bits, all loss bits are recovered by some help of the inner-edges. In Table 2.1 and Table 2.3, for example, we consider in the case that column 4 and 5 are survivors, the structure of graph is shown in Fig.2.5.

Clearly, we can see that a Hamiltonian cycle is constructed from node 21, edges of $0 + 14$, $2 + 23$, and $9 + 16$ by an assistance of all inner-ring edges (dotted lines): $0+1$, $1+2$, $7+8$, $8+9$, $14+15$, $15+16$, $21+22$, and $22+23$,

Table 2.2: B_1

-	1	-	-	-	-	-
8	-	-	-	-	-	-
-	-	-	-	-	-	15
-	-	-	-	-	22	-
-	-	-	-	-	-	-
-	-	-	-	-	-	-
-	-	-	-	-	-	-
-	-	-	-	-	-	-
-	-	-	-	-	-	1+8
-	-	-	-	-	1+15	-
-	-	1+22	-	-	-	-
-	-	8+15	-	-	-	-
-	8+22	-	-	-	-	-
15+22	-	-	-	-	-	-

which are connected via middle ring (or layer) between them. In this case, a Hamiltonian cycle is always occurred to connect all nodes in a CGR graph.

To make a clearer view of CGR structure, we will relabel all nodes as $V(K_4, C_7)$, where K_4 represents a vertex in a complete graph ith of K_4 which $i = 0, 1, \cdots, 6$, and C_7 represents a vertex that has a connection in the same ring jth which $j = 0, 1, \cdots, 4$. For example, node 0 is denoted by $V(0, 0)$, and node 1 is denoted by $V(1, 0)$.

Table 2.3: B_2

-	-	2	-	-	-	-
-	9	-	-	-	-	-
16	-	-	-	-	-	-
-	-	-	-	-	-	23
-	-	-	-	-	-	-
-	-	-	-	-	-	-
-	-	-	-	-	-	-
-	-	-	-	-	-	-
2+9	-	-	-	-	-	-
-	-	-	-	-	-	2+16
-	-	-	2+23	-	-	-
-	-	-	9+16	-	-	-
-	-	9+23	-	-	-	-
-	16+23	-	-	-	-	-

2.3.2 Perfect One-Factorization (P1F) as the Inter-Ring Edges Shifting Index Assigning Algorithm

In this section, we will describe how we use one of the known P1F technique to label the base graph. By definition, a *one-factorization* of a graph is a partitioning of the set of its edges into subsets such that each subset is a graph of degree one [12]. A *perfect one-factorization* is a particular one-factorization in which the union of any pair of one-factors forms a Hamiltonian cycle.

Remark 2.3.1. A Hamiltonian cycle is a cycle in an undirected graph which visits each vertex exactly and only once and also returns to the starting one.

For a base graph K_{v_1}, we assign vertex number as $0, 1, 2, \cdots, v_2$. Then we add two more vertices denoted by $-\infty, +\infty$ in to the base graph. Now, the graph becomes K_{v_1+2}. Next, draw a complete graph in a cycle form having the

Table 2.4: B_3

-	-	-	3	-	-	-
-	-	10	-	-	-	-
-	17	-	-	-	-	-
24	-	-	-	-	-	-
-	-	-	-	-	-	-
-	-	-	-	-	-	-
-	-	-	-	-	-	-
-	-	-	-	-	-	-
-	3+10	-	-	-	-	-
3+17	-	-	-	-	-	-
-	-	-	-	3+24	-	-
-	-	-	-	10+17	-	-
-	-	-	10+24	-	-	-
-	-	17+24	-	-	-	-

vertex $-\infty$ at the center, Fig. 2.3 demonstrates the case of K_4 with 2 extra vertices added.

Then, we label the edges into v_1+1 sets, each set consists of

$$E_i = \{(-\infty, i)\}$$

and edges that are diagonal to the $(-\infty, i)$. The following table shows the labeling result from Fig 2.3.

Table 2.5: B_4

-	-	-	-	4	-	-
-	-	-	11	-	-	-
-	-	18	-	-	-	-
-	25	-	-	-	-	-
-	-	-	-	-	-	-
-	-	-	-	-	-	-
-	-	-	-	-	-	-
-	-	-	-	-	-	-
-	-	4+11	-	-	-	-
-	4+18	-	-	-	-	-
-	-	-	-	-	4+25	-
-	-	-	-	-	11+18	-
-	-	-	-	11+25	-	-
-	-	-	18+25	-	-	-

set	edges
$(-\infty, +\infty)$	$(0,3)$, $(1,2)$
$(-\infty, 0)$	$(1,+\infty)$,$(2,3)$
$(-\infty, 1)$	$(0,2)$,$(3,+\infty)$
$(-\infty, 2)$	$(1,3)$,$(0,+\infty)$
$(-\infty, 3)$	$(0,1)$,$(2,+\infty)$

Then, remove all edges connected to $-\infty$ and $+\infty$, we have the following edges left as shown in Fig. 2.6 and the table below.

Table 2.6: B_5

-	-	-	-	-	5	-
-	-	-	-	12	-	-
-	-	-	19	-	-	-
-	-	26	-	-	-	-
-	-	-	-	-	-	-
-	-	-	-	-	-	-
-	-	-	-	-	-	-
-	-	-	-	-	-	-
-	-	-	5+12	-	-	-
-	-	5+19	-	-	-	-
-	-	-	-	-	-	5+26
-	-	-	-	-	-	12+19
-	-	-	-	-	12+26	-
-	-	-	-	19+26	-	-

set	edges
$(-\infty,+\infty)$	$(0,3)$, $(1,2)$
$(-\infty,0)$	$(2,3)$
$(-\infty,1)$	$(0,2)$
$(-\infty,2)$	$(1,3)$
$(-\infty,3)$	$(0,1)$

Next, we can label edges in each group by using the following rules.

1. Edges in the $(-\infty,+\infty)$ are label as v_1+2.
2. Edges in the other sets can be labelled any number from $0, 1, \cdots, v_1 - 1$.
3. For any edge, the number labelled must not be equal to the vertex number at both ends of the edge.

Table 2.7: B_6

-	-	-	-	-	-	6
-	-	-	-	-	13	-
-	-	-	-	24	-	-
-	-	-	27	-	-	-
-	-	-	-	-	-	-
-	-	-	-	-	-	-
-	-	-	-	-	-	-
-	-	-	-	-	-	-
-	-	-	-	6+13	-	-
-	-	-	6+20	-	-	-
6+27	-	-	-	-	-	-
13+20	-	-	-	-	-	-
-	-	-	-	-	-	13+27
-	-	-	-	-	20+27	-

So we can have several possibilities of this labeling, for example, like the tables shown below.

set	edges	label
$(-\infty, +\infty)$	$(0, 3)$, $(1, 2)$	6
$(-\infty, 0)$	$(2, 3)$	1
$(-\infty, 1)$	$(0, 2)$	3
$(-\infty, 2)$	$(1, 3)$	0
$(-\infty, 3)$	$(0, 1)$	2

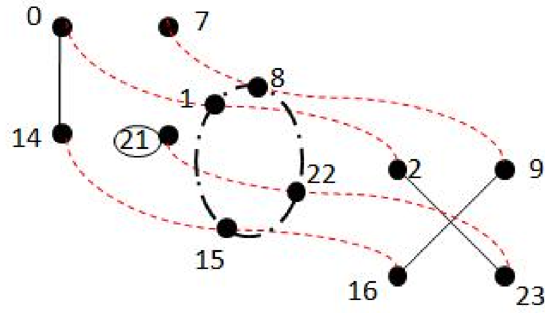


Figure 2.5: A Hamiltonian cycle formed by 2 survivors of (K_4, C_7)

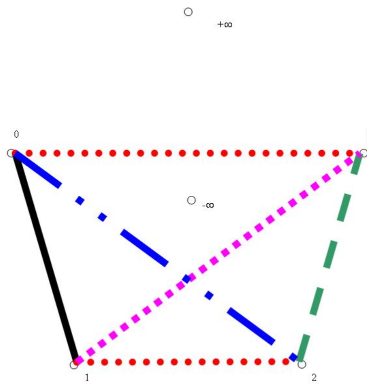


Figure 2.6: Complete graph K_4 after trimming K_6 .

set	edges	label
$(-\infty, +\infty)$	$(0,3)$, $(1,2)$	6
$(-\infty, 0)$	$(2,3)$	0
$(-\infty, 1)$	$(0,2)$	1
$(-\infty, 2)$	$(1,3)$	2
$(-\infty, 3)$	$(0,1)$	3

According to the numbers labelled on all edges of the base graph K_{v_1}, we suppose that an edge (i, j) in the base graph is labelled with x, then a row in

an array code containing all (inter-ring) edges connecting between ring i and j is shifted by x.

2.4 Dual CGR Codes

To construct the dual code of a CGR code is simple, and can be achieved by swaping the duty of vertices and edges in the CGR graph. Now let each edge represent an information bit, and each vertex represent a parity bit by XORing information bits depended on a degree of each vertex. So we can rearrange CGR(K_{v_1}, C_{v_2}) in an array of $\frac{v_1 v_2}{2} \times v_2$. From the previous example of CGR(K_2, C_5), we will get its dual code as shown in the below example.

Example 4: The (5×5) array code in Example 2 is a $(5, 2)$ 3-erasure-correcting MDS code constructed from CGR(K_2, C_5) in Fig. 2.2. The same array arrangement can be mapped to a dual MDS code with 2-erasure-correcting capability, by reversing the roles of edges and vertices (i.e. letting edges represent data bits and vertices represent parity bits). To ease the representation, we re-label the edges using alphabets $a, b, ...o$ as in Fig. 2.2, and interpret the vertices of degree 3 as parities on 3 information bits. The $(5, 3)$ dual code takes the following form:

$a \oplus l \oplus e$	$a \oplus m \oplus b$	$b \oplus n \oplus c$	$c \oplus o \oplus d$	$d \oplus k \oplus e$
$m \oplus g \oplus h$	$h \oplus n \oplus i$	$o \oplus i \oplus j$	$j \oplus k \oplus f$	$g \oplus l \oplus f$
c	d	e	a	b
i	j	f	g	h
k	l	m	n	o

This dual code is also an MDS code where its proof is shown as follows.

2.4.1 Proofs of Duality of CGR Codes

This section presents all proofs of duality of codes constructing based on CGR graph. From one CGR graph, we can construct two codes, the CGR code and its dual. The original CGR code is constructed by viewing vertices as systematic bits and edges as parity bits. Its dual code uses vertices as parity bits while edges as information bits. We show that the CGR code and its dual are dual codes of each other, e.g. H matrix of the code is equivalent to G matrix of the dual, and vice versa.

Definition 2.4.1. Consider a graph $G(V, E)$, we define construction methods for the CGR code and its dual from the graph as follows.

CGR code construction

1. For each node $v \in V$, v represents an information bit.
2. For each edge $e \in E$, e represents a parity bit.
3. By grouping these nodes and edges into a set of symbols, we obtain S_i where S_i is an ith symbol consisting of r edges and k nodes.
4. A set of symbols S_i forms a CGR code.

CGR dual code construction

1. For each node $v \in V$, v represents an parity bit.
2. For each edge $e \in E$, e represents a information bit.

3. By grouping these nodes and edges into a set of symbols, we obtain S_i where S_i is an ith symbol consisting of r edges and k nodes.

4. A set of symbols S_i forms a CGR code.

The H matrix of the code will be in the form of block matrix, where each column corresponds to a symbol S_i.

$$H \equiv \begin{bmatrix} h_{1,1} & h_{1,2} & .. & h_{1,m} \\ h_{2,1} & .. & .. & .. \\ .. & .. & .. & .. \\ h_{n,1} & .. & .. & h_{n,m} \end{bmatrix}$$

where

$$h_{i,j} = \begin{Bmatrix} [\bar{0}\, I] & ,(i = j) \\ [P_{i,j}\, \bar{0}] & ,(i \neq j) \end{Bmatrix}$$

We can define G as follow.

$$G \equiv \begin{bmatrix} g_{1,1} & g_{1,2} & .. & g_{1,n} \\ g_{2,1} & .. & .. & .. \\ .. & .. & .. & .. \\ g_{m,1} & .. & .. & g_{m,n} \end{bmatrix},$$

where

$$g_{i,j} = \begin{Bmatrix} [I\, \bar{0}] & ,(i = j) \\ [\bar{0}\, P^T_{i,j}] & ,(i \neq j) \end{Bmatrix}$$

Proposition 2.4.2. If HG^T is in the form $\begin{bmatrix} P & I \end{bmatrix} \begin{bmatrix} I \\ P \end{bmatrix}$ and HG^T is valid under multiplication, then $HG^T = 0$.

Lemma 2.4.3. For the matrix H and G of CGR code,

$$HG^T = 0$$

Proof. Consider the matrix H of CGR code,

$$H \equiv \begin{bmatrix} h_{1,1} & h_{1,2} & .. & h_{1,m} \\ h_{2,1} & .. & .. & .. \\ .. & .. & .. & .. \\ h_{n,1} & .. & .. & h_{n,m} \end{bmatrix}$$

we can perform column-wise re-ordering by separating each element of $[\bar{0}\, I]$ and $[P_{i,j}\, \bar{0}]$ into 2 groups as shown below.

$$H \equiv \begin{bmatrix} P_{1,1} & P_{1,2} & .. & P_{1,m} & I & 0 & 0 & 0 \\ P_{2,1} & .. & .. & .. & 0 & I & 0 & 0 \\ .. & .. & .. & .. & 0 & 0 & I & 0 \\ P_{n,1} & .. & .. & P_{n,m} & 0 & 0 & 0 & I \end{bmatrix}$$

Exact transformation can be applied to G.

$$G \equiv \begin{bmatrix} I & 0 & 0 & 0 & P^T_{1,1} & P^T_{1,2} & .. & P^T_{1,n} \\ 0 & I & 0 & 0 & P^T_{2,1} & .. & .. & .. \\ 0 & 0 & I & 0 & .. & .. & .. & .. \\ 0 & 0 & 0 & I & P^T_{m,1} & .. & .. & P^T_{m,n} \end{bmatrix}$$

Assume $\bar{P} = \begin{bmatrix} P_{1,1} & P_{1,2} & .. & P_{1,m} \\ P_{2,1} & .. & .. & .. \\ .. & .. & .. & .. \\ P_{n,1} & .. & .. & P_{n,m} \end{bmatrix}$, then we can write $HG^T = \begin{bmatrix} \bar{P} & I \end{bmatrix} \begin{bmatrix} I \\ \bar{P} \end{bmatrix}$.

From Proposition 2.4.2, $HG^T = \begin{bmatrix} \bar{P} & I \end{bmatrix} \begin{bmatrix} I \\ \bar{P} \end{bmatrix} = 0$ □

Theorem 2.4.4. Given a graph $G(V, E)$, two codes, that are constructed with different approaches A and B, are dual codes of each other, e.g. H of the original code is G of the dual code.

Proof. From the construction of H of the code, each row in $\bar{P}$ represents two nodes connecting an edge as in Fig. 2.7.

Now consider H matrix of the dual code, from the definition of graph construction, each row in $\bar{P}'$ represents a node connecting edges as in Fig. 2.8.

If $\bar{P}$ and $\bar{P}'$ are produced from the same graph, then $\bar{P}' = \bar{P}^T$, hence H of the dual code equals to G of the original code, because G of the original code is in form of $\bar{P}^T$. The same prove can be applied where H of original code equals to G of the dual code. □

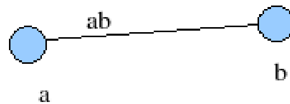


Figure 2.7: Graph representing a row in H

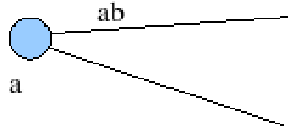


Figure 2.8: Graph representing a row in H of the dual code

2.5 Connection to B-Codes

The CGR code has some similarity to the B-codes in both of graph structure and data layout algorithm. This is due to the fact that our code consists of complete-graph-like structure as well as the labeling algorithm that used for data array arrangement.

The new code, however, has different properties/parameters. The connection between the CGR code and B-codes in [3] can be viewed via a transformation process that shorten the CGR code to B-codes. The basic idea behind this connection is that when we view in another way, e.g. the CGR graph as a graph consisting of multiple layers, each layer forms a single complete graph. These sub layers are connected via the edges $E_{(i,j),\,(i+1) \mod K,j}$ for all K. Obviously, each individual layer can be used to form a B-Code. Hence, decomposing CGR code will result in B-Code, or, B-code can be considered as a reduced form of CGR code. Inversely, connecting complete graphs together as a super ring forms a CGR code. Data layering procedure helps glueing this code to be one MDS code by shifting each layer before placing each vertex element into disk array.

Another notable difference between CGR codes and B-Codes is the symmetry in the array structure. The B-code B_{2n+1} constructed from K_{2n} graph has the structure shown in Fig. 2.10 (b), which is not symmetric in the number of information bits versus the number of parity bits per column, while the

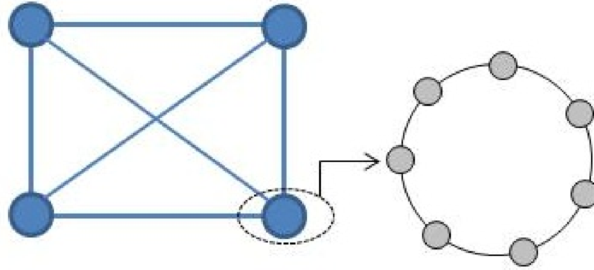


Figure 2.9: A super graph represents a CGR(K_4, C_7) code, where each super node has 7 nodes and there are 7 edges represented in each inter-edge.

CGR code constructed from a similar K_{2n} super graph is in the symmetric form which is shown in Fig. 2.10 (a).

Example 5: Consider a CGR code and a B-code constructed from K_{2n} graph where $n = 2$. Fig. 2.9 illustrates the CGR(K_4, C_7) code constructed from a complete graph K_4 and 4 ring graphs of 7 nodes each. The code can be written in the array form as follows.

0	1	2	3	4	5	6
8	9	10	11	12	13	7
16	17	18	19	20	14	15
24	25	26	27	21	22	23
4+5	5+6	6+0	0+1	1+2	2+3	3+4
11+12	12+13	13+7	7+8	8+9	9+10	10+11
18+19	19+20	20+14	14+15	15+16	16+17	17+18
25+26	26+27	27+21	21+22	22+23	23+24	24+25
2+9	3+10	4+11	5+12	6+13	0+7	1+8
3+17	4+18	5+19	6+20	0+14	1+15	2+16
6+27	0+21	1+22	2+23	3+24	4+25	5+26
13+20	7+14	8+15	9+16	10+17	11+18	12+19
7+21	8+22	9+23	10+24	11+25	12+26	13+27
15+22	16+23	17+24	18+25	19+26	20+27	14+21

Note that each vertex denoting the data is labeled by a number from $0-27$. Each row possesses a shifted cyclic symmetry.

Now, consider only the first vertex of each ring and all the edges connected between the selected vertices. The other vertices and edges are punctured as below.

0	-	-	-	-	-	-
-	-	-	-	-	-	7
-	-	-	-	-	14	-
-	-	-	-	21	-	-
-	-	-	-	-	-	-
-	-	-	-	-	-	-
-	-	-	-	-	-	-
-	-	-	-	-	-	-
-	-	-	-	-	0+7	-
-	-	-	-	0+14	-	-
-	0+21	-	-	-	-	-
-	7+14	-	-	-	-	-
7+21	-	-	-	-	-	-
-	-	-	-	-	-	14+21

Then, vertically and horizontally compact and rewrite the new array as follows.

Horizontal compacting:

0	0+21	-	-	21	14	7
7+21	7+14	-	-	0+14	0+7	14+21

Vertical compacting:

0	0+21	21	14	7
7+21	7+14	0+14	0+7	14+21

Reordering:

0	7	14	21	0+21
7+21	14+21	0+7	0+14	7+14

This essentially leads to a B_{2n+1} code.

a1	a2	a3	a4	a1+a3
a2+a3	a3+a4	a4+a1	a1+a2	a2+a4

Thus, we have shown the shortening procedure that slices part of the CGR code and maps it to a B_{2n+1} code. From this transformation, we can conclude that B-codes are degeneration of CGR codes.

2.5.1 Discussion

We now discuss and analyze the properties of our CGR codes. CGR graphs are nested graphs with a complete graph as the base graph, and B-codes can be constructed from complete graphs. Hence, it should not be surprising that CGR codes subsume B-codes as contracted codes. However, in addition to the significantly more complex structure of CGR codes, another notable difference is that CGR codes are by nature cyclically symmetric, whereas B_{2n+1} codes have an asymmetric structure as shown in Fig. 2.10.

We consider the complexity of CGR codes.

Definition 2.5.1. The *update complexity* is defined as the number of parity updates required while a single information bit is changed or updated, averaged over all the information (systematic) bits [13].

Information
Inner-edge parity
Inter-edge parity

Information | Parity
Parity

(a) CGR code structure (b) B_{2n+1} code structure

Figure 2.10: (a) Structure of CGR code. (b) Structure of B_{2n+1} code

Definition 2.5.2. The *decoding complexity* is defined as the number of bit operations (e.g. XOR, AND, shift) required in order to recover the erased symbols (columns) from the survivors, averaged over all the information symbols.

Recall that the proposed CGR codes based on K_{v_1} and C_{v_2} where $v_2 = v_1 + 3$. For updating a single information bit, since every information bits involves $v_1 + 1$ parity bits, it will give rise to the update of $(v_1 + 1)$ parity bits. Hence, averaged over $v_1 v_2$ information bits, the update complexity will be $\frac{v_2 - 2}{v_2(v_2 - 3)}$. The update complexity for different code configurations is listed in Table 2.8. Since the code is one with parameters $(n, k) = (v_2, 2)$, the update complexity decreases linearly with the "code-length" v_2 and goes to zero asymptotically.

To compute the decoding complexity, we can consider that all the $\frac{v_1 v_2}{2}$ bits in an arbitrary missing column takes one XOR operation per bit, or, one XOR operation for the entire symbol. In the worst case, the code has a payload of two systematic symbols (or $v_1 v_2$ systematic bits), so the decoding complexity is $\frac{1}{2}$ per erased symbol, irrespective of the code lengths.

K_{v_1}	C_{v_2}	code	update complexity	decode complexity
K_2	C_5	(5,2)	$3/10 = 30\%$	$1/2 = 50\%$
K_4	C_7	(7,2)	$5/28 = 18\%$	$1/2 = 50\%$
K_6	C_9	(9,2)	$7/54 = 13\%$	$1/2 = 50\%$
K_8	C_{11}	(11,2)	$9/88 = 10\%$	$1/2 = 50\%$
K_{10}	C_{13}	(13,2)	$11/130 = 8\%$	$1/2 = 50\%$

Table 2.8: Update complexity and decoding complexity

In conclusion, our main contributions are the systematic code representation and algorithmic construction based on a special type of graph called the complete-graph-of-rings graph. The P1F is used as a tool for labeling and mapping graphs to array codes. This code achieves the MDS property and is an optimal code. The dual code is also discussed and is an MDS code. We have also shown the direct relation of the new code to the B-code and provided the transformation scheme that shortens the new code to B-code. Even though the code has shown its elegant well-structured code representation, however, the code rate is relatively low and approaching zero as the erasure recovering capability (the number of redundancy), increases.

2.6 Low-Density MDS Array Codes

Any block code can be described by a generator matrix (G) and parity-check matrix (H) [10], [11]. In this section, we consider representing CGR codes in terms of matrices, and draw connection to LDPC codes. Note that an array code is considered as a *low density* code if it has the smallest possible update complexity for its parameters [15].

Since from the previous section, we have introduced and generated a CGR code based on a complete-graph-of-ring graph, here we will investigate an MDS array code based on other graphs and show that it is a low-density MDS array code. Such a new code construction based on graphs which we assign and place all vertices and edges into an appropriate array size aims to achieve the strong MDS property of an MDS array code which is defined in Definition 2.6.1:

Definition 2.6.1. An array code of size $m \times n$ with mk information bits and $(n-k)m$ parity bits has the MDS property, if, for any given r columns and any given series of positive integers a_i, where $k \leq r \leq n$, $0 \leq a_i \leq m$, $1 \leq i \leq r$, and $\sum_{i=1}^{r} a_i = mk$, there always exist a_i bits from the ith column such that the mk information bit can be recovered from these mk bits from the r columns.

To describe an MDS array code in the matrix form, the array code of size $m \times n$ arranged in an array of m rows and n columns with mk information bits and $(n-k)m$ parity bits. Then, this code has a code rate of $\frac{k}{n}$. Each row is defined as a *strip* and each column contains an $n-$part encoded symbols and is viewed as one disk. If this $(m \times n)$ array code guarantees t fault tolerance, it means that after losing any set of t columns, the remainder $n-t$ columns are sufficient to recover all symbols or all loss columns.

Let G be a generator matrix of size $km \times nm$ and H be a parity-check matrix of size $(n-k)m \times nm$. An example of a 2×4 array code is shown in both the array form and the matrix. For a given information sequence x of length km, the codeword y still holds $y = xG$ and $yH^T = 0$, where $GH^T = 0$.

a	b	c	d
$b \oplus c$	$c \oplus d$	$d \oplus a$	$a \oplus b$

With $n = 4, m = 2, k = 2$, its parity-check matrix can be described as follows.

$$H = \left[\begin{array}{cc|cc|cc|cc} 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \end{array}\right]$$

Accordingly, its generator matrix is as follows.

$$G = \left[\begin{array}{cc|cc|cc|cc} 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \end{array}\right]$$

This (2×4) array code has a code rate of $1/2$ and tolerate $t = 2$ erasures. Additionally, we can consider the dual of array codes as the one for a one-dimensional linear block code as a definition given below.

Definition 2.6.2. Let C be a linear array code of size $n \times m$ over $GF(q)$, then its *dual code* $C^{\perp}$ is defined as $C^{\perp} = \{u \in GF(q)^{nm} : u \cdot v = 0 \text{ for all } v \in C\}$, where $\cdot$ is the conventional dot product of vector.

Naturally followed by the given definition of dual codes, the parity-check matrix of an array code is the generator matrix of its dual code.

2.6.1 Low-Density CGR Codes

We now illustrate CGR code in matrices including both parity-check matrix and generator matrix, which are both sparse and systematic. We show that this

code is not only in a sparse code, but also is in a systematic form. The parity-check matrix and generator matrix shown below are from the CGR(K_2, C_5) code whose layout is illustrated in Table 2.9. In the previous section, this code and its dual are proved and expressed as MDS codes that can tolerate 3 and 2 erasures, respectively.

$$H = \left[\begin{array}{ccccc|ccccc|ccccc|ccccc|ccccc}
0&0&1&0&0&0&0&0&0&0&1&0&0&0&0&1&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&1&0&0&0&0&1&0&0&0&0\\
1&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&1&0&0&0&0\\
1&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0\\
0&0&0&0&0&1&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&1&0&0\\
0&0&0&1&0&0&1&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&1&0&0&1&0&0&0&0&1&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&1&0&0&0&0&1&0&0&0\\
0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&1&0&0&0\\
0&1&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0\\
0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&1&0&0&0&0\\
1&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0\\
0&1&0&0&0&1&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&1&0&0&0&1&0&0&0&0&0&0&0&0&1&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&1&0&0&0&0&0&0&0&0&1
\end{array}\right]$$

Since $GH^T = 0$, its generator matrix, G is as follow.

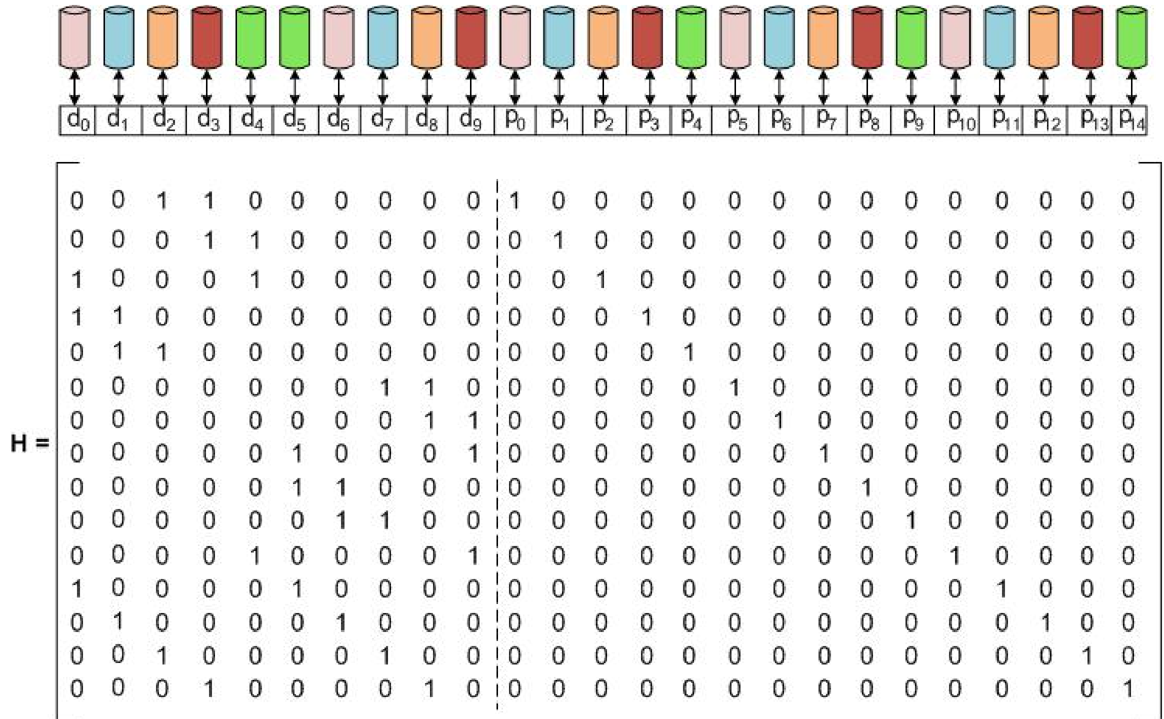
$$
G = \left[\begin{array}{ccccc|ccccc|ccccc|ccccc|ccccc}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}\right]
$$

This CGR(K_2, C_5) code, which is the $(m \times n) = (5 \times 5)$ array code, has the parity-check matrix of size $(mk \times nm) = (15 \times 25)$, and the generator matrix of size $((n-k)m \times nm) = (10 \times 25)$, where $k = 2$. This code can tolerate triple disk failures, while its dual code has double disk failures toleration.

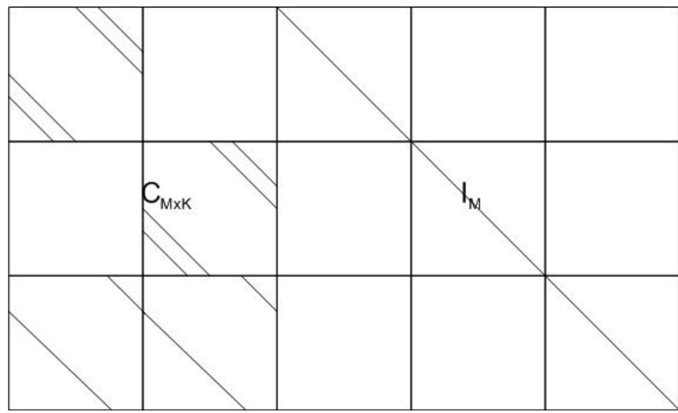
$\mathbf{d_0}$	d_1	d_2	d_3	d_4
d_6	d_7	d_8	d_9	$\mathbf{d_5}$
$p_0 = 2 \oplus 3$	$p_1 = 3 \oplus 4$	$p_2 = 4 \oplus 0$	$\mathbf{p_3 = 0 \oplus 1}$	$p_4 = 1 \oplus 2$
$p_5 = 7 \oplus 8$	$p_6 = 8 \oplus 9$	$p_7 = 9 \oplus 5$	$\mathbf{p_8 = 5 \oplus 6}$	$p_9 = 6 \oplus 7$
$p_{10} = 4 \oplus 9$	$\mathbf{p_{11} = 0 \oplus 5}$	$p_{12} = 1 \oplus 6$	$p_{13} = 2 \oplus 7$	$p_{14} = 3 \oplus 8$

Table 2.9: The array CGR(K_2, C_5) code

To show that the code is systematic, the parity check matrix can also be rewritten in Fig. 2.11(a). This new parity check matrix takes the following systematic form as:



(a) The relation between a systematic parity-check matrix and disk array of an CGR(K_2, C_5) code



(b) A systematic parity-check matrix of an CGR(K_2, C_5) code

Figure 2.11: The parity check matrix of CGR(K_2, C_5) array code

$$H = \left[\begin{array}{c|c} C_{M\times K} & I_M \end{array}\right] = \left[\begin{array}{c|c|c|c|c} C_m^3 & 0_m & I_m & 0_m & 0_m \\ \hline 0_m & C_m^3 & 0_m & I_m & 0_m \\ \hline C_m^2 & C_m^2 & 0_m & 0_m & I_m \end{array}\right], \tag{2.1}$$

where $C_{M\times K}$ is an $M \times K = 15 \times 10$ quasi-cyclic matrix and I_M is an $M \times M = 15 \times 15$ identity matrix which is a combination of identity sub matrices of size $m \times m = 5 \times 5$. The cyclic matrix $C_{M\times K}$ is also separated into 6 sub matrices with size $m \times m = 5 \times 5$ each. Note that C_m^3 and C_m^2, are cyclic squares with 2 and 1 diagonals and are 3- and 2-bits left cyclically shifted, respectively, and 0_m is the zero matrix of size $m \times m$. We can consider the codeword as $c = (d_0, d_1, ..., d_{K-1}, p_0, p_1, ..., p_{M-1})$, where d_i is an information bit ($i \in 0, 1, ..., K-1$), and p_j is a check bit ($j \in 0, 1, ..., M-1$).

Therefore, for general array sizes of $\left(\frac{k(k+3)}{2} \times (k+3)\right)$ CGR codes, where k is an even integer of information-bit rows and $k \geq 2$, this code over $GF(2)$ can be defined by the following parity-check matrix:

$$H := \left[\begin{array}{c|c|c|c|c|c|c} C_{k+3}^{s_1} & 0_{k+3} & ... & 0_{k+3} & I_{k+3} & 0_{k+3} & 0_{k+3} \\ \hline 0_{k+3} & C_{k+3}^{s_2} & 0_{k+3} & .... & 0_{k+3} & I_{k+3} & 0_{k+3} \\ \hline \vdots & \ddots & \ddots & \vdots & \vdots & \ddots & \vdots \\ \hline C_{k+3}^{s_M} & C_{k+3}^{s_M} & ... & C_{k+3}^{s_M} & 0_{k+3} & ... & I_{k+3} \end{array}\right], \tag{2.2}$$

Note that the size of this parity-check matrix is M-by-$(K + M)$, which equals to $(k\frac{k+1}{2})$-by-$(\frac{k(k+3)^2}{2})$, and s_i is the offset vector of left-cyclically shifting of row ith.

Theorem 2.6.3. Consider the parity-check matrix (H) of CGR (K_{v_1}, C_{v_2}) MDS array code of size $\dfrac{k(k+1)}{2} \times \dfrac{k(k+3)^2}{2}$, where k is the number of data disks (or nodes in CGR graph) and $k = v_1 v_2$, its column weight (w_c) equals to the degree of each node, $w_c = v_1 + 1$, and its row weight (w_r) is 3.

Proof. From the structure of CGR graph, each parity (an edge in CGR graph) is computed by XORing 2 data bits (nodes in CGR graph), so in each row of H, when the parity columns are represented in an identity matrix, the row weight (w_r) is 3.

Then, the column of systematic bits in H is a part of a group of quasi-cyclic square matrices, which correspond to the nodes of CGR graph. To place all nodes into an array, they will first appear as data disks, and they will also appear to construct parity disks. Thus, their appearances are depended on the degree of nodes which equals to $v_1 + 1$ resulting in the column weight of H, $w_c = v_1 + 1$. □

2.6.2 Data Recovery via Parity-Check Matrix

We consider storage systems when t disks are lost or deleted, and therefore t columns of an $(m \times n)$ MDS array code has been deleted. Thus, the original parity-check matrix of size $(M \times (K + M))$, where $M = (n - k)m$. t erasures will affect not only t rows, but they also delete all rows and columns that are represented symbols stored in the same disk.

Now, the systematic parity-check matrix of erasure correcting code after t disks are failed/loss is expressed as follows:

$$H_{(n-k)(m-t)\times n(m-t)} = \left[\ C_{(n-k)(m-t)\times(n-k)t} \,\middle|\, I_{(n-k)(m-t)}\ \right],$$

where $C_{(n-k)(m-t)\times(n-k)t}$ is an $(n-k)(m-t)\times(n-k)t$ left-cyclically shift matrix (followed by the offset vector), and $I_{(n-k)(m-t)}$ is the $(n-k)(m-t)\times(n-k)(m-t)$ identity matrix. So, to recover the disk failures, we define H^t as a parity-check matrix of t disk failures with the size shown above.

Theorem 2.6.4. A $(n-k)m\times nm$ parity-check matrix (H) of an $(n\times m)$ CGR MDS array codes contained km informations bits and $(n-k)m$ parity bits will be reduced into a parity-check matrix H^t of size $(n-k)(m-t)\times n(m-t)$ after there are t disk failures.

Proof. The construction of CGR codes is based on the connection of CGR graphs where the number of data nodes in each ring is $k+3$ from a total k rings, and their edges represent parity bits computed by XORing two nodes at both ends, then we place both data bits and parity bits into an array of size $n\times m$. This code is a vertical array code, so that each column contains both data and parity bits. When there are t erasures, we will have only $m-t$ survivors to recover all loss data (note that we have already proved that the survivors can recover all failures back if $t\leq d-1$), so the new array code of size $n\times(m-t)$ will also construct the new parity-check matrix of size $(n-k)(m-t)\times n(m-t)$. □

Here the first $(n-k)t$ columns in H^t correspond to information bits in the survivor disks and the last $(n-k)(m-t)$ columns correspond to the parity bits of survivors. Fig.2.12 and Fig.2.13 shows the example of row-operation and column-operation decoding of 3-disk failures of an CGR(K_2, C_5) code where its original parity-check matrix is shown in Fig. 2.11(a). Note that all red columns is labeled as the loss information and parity bits in all disk failures.

To recover all loss data, we start with the row-operation decoding as shown

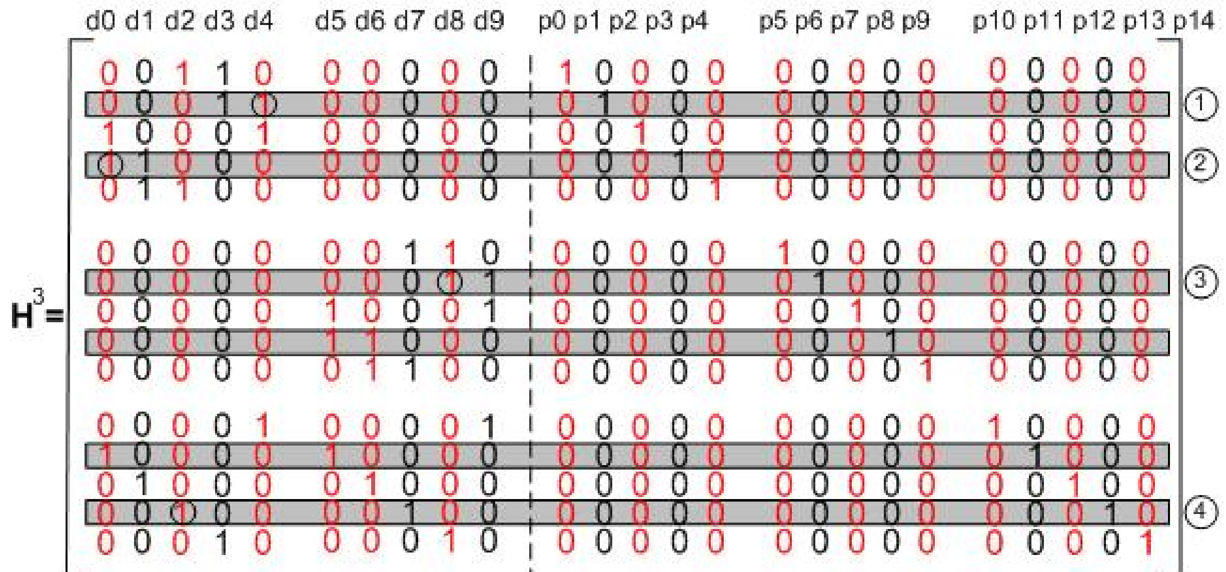


Figure 2.12: A row-decoding process of the H matrix of $CGR(K_2, C_5)$ code

in Fig.2.12. We first check the parity columns in order as shown in the number (in the circle) next to the grey line and the bit which is in circle is now recovered by XORing its associated survivor data and parity bits. Then, if there are not enough survivor data bits in the same row of such survivor parity bit, we move to the column-operation decoding. Illustrated in Fig. 2.13, the process is in order shown next to the columns and data bits which are illustrated in squares, stars, and triangular are recovered.

Therefore, we can summarize the decoding algorithm, namely "*the row- and column-operation decoding algorithm*," for the parity-check matrix H^t after the t disk failures occur.

Row- and Column-Operation Decoding Algorithm

1. In the row ith of a survivor parity-check matrix, check the associated data bits of this parity if they are lost or not. If there is only one

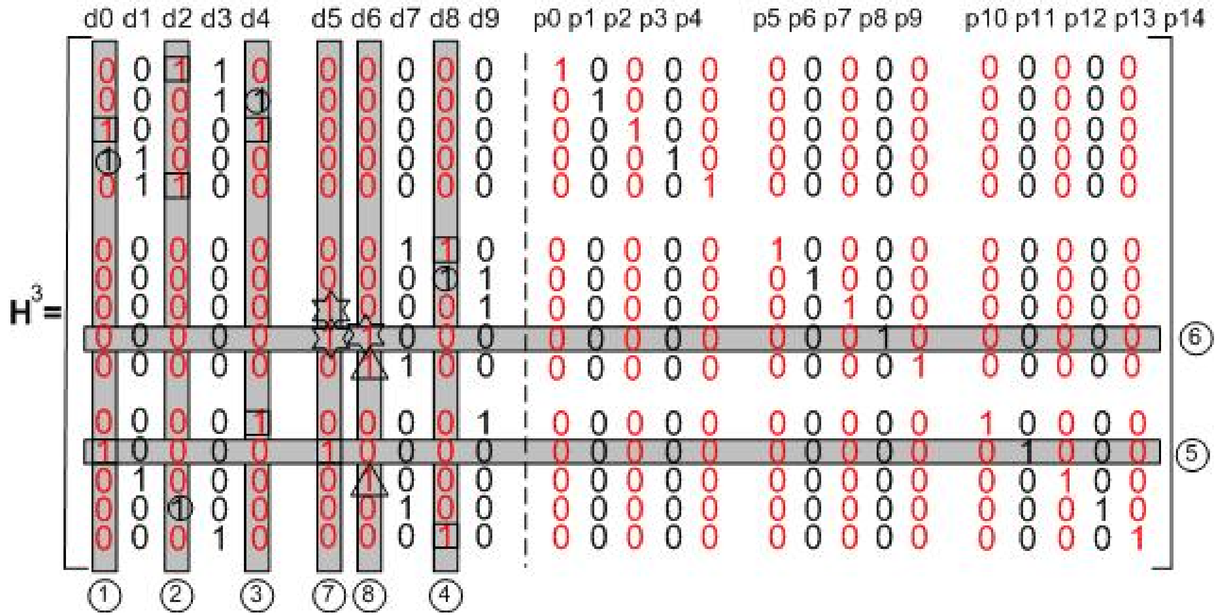


Figure 2.13: A column-decoding process of the H matrix of $CGR(K_2, C_5)$ code

associated data bit loss, recover it by XORing this parity bit with other associated data bits. If there are not enough survivor data bits, go and take a process at the column-operation decoding.

2. In the column jth of an associated information bit, (i) check the data bit of the disk that may recover from the row ith, (ii) at the jth column, and the iith row, where $ii \neq i$, XORing this data bit with the parity bit of this row again, (iii) stop when all data and parity bits are recovered.

Unlike traditional LDPC codes, there are correlations between some columns of the parity check matrix, since they correspond to the same disk. When the disk fails, all data and parity bits stored in these strips are lost. Thus, the set of strips in the same disk will be either alive or dead at the same time. Array codes differ from LDPC codes by adding restrictions to their code graph based on array column locations.

In the future, we can exploit the parity check matrix to shed insight into constructing new MDS codes.

Chapter 3

Nested codes with Hierarchical protection for distributed storage networks

In this chapter we introduce the erasure-correcting code which can provide the erasure protection and fault tolerance for a large-scale storage network or a data center with a huge number of disks. Inspired by the concept of hierarchical protection, we firstly consider code schemes that "nest" MDS optimal code with LT codes, and evaluate their trade off and complexity. We next propose an erasure coding scheme based on a rigid structure of MDS local codes wrapped by single parity check (SPC) codes in both horizontal and vertical dimensions.

3.1 Introduction

Data centers, or, distributed storage networks are on the high rise, where hundreds of or thousands of storage nodes are pulled together to provide massive storage capacity. Each storage node may be physically implemented by a cheap computer, each of which consists of and controls an array of, say, ten, hard disks like the RAID's systems. As the system grows, the chance of component failures and the number of disk failures also increase, so efficient techniques to protect against data loss become more important and necessary.

In the previous Chapter, we have proposed the CGR codes [23, 24] which are based on graph structure in order to handle disk failures in disk arrays. In this chapter, we extend these codes (and other existing MDS or near-MDS codes) to achieve better failure protection and handle a larger number of failures for large storage networks. The new scheme we have developed, termed "*nested codes with hierarchical protection for distributed storage networks*", target large storage systems. Since the distributed storage network is large, a general challenge is to provide data consistency while preventing failures and allowing concurrent access [41]. Our goal is to achieve highly-scalable, space-efficient and access-efficient (n, k) MDS codes, where $n - k \leq k$, that is, the number of redundant (spare) nodes is no greater than the number of data nodes.

In the context of error-correcting codes or erasure-correcting codes for large storage systems, *GRID codes* [39] are one good example of multidimensional codes in distributed disk arrays. This class of codes are completely XOR-based, non-MDS codes which can tolerate up to 15 and higher disk failure. They use a concept of *matched codes* (a group of codes that are chosen to construct a GRID code) to construct a GRID code. This code has a very regular structure, which is constructed by a simple grid of rows and columns, denoted

by GRID($code_r, code_c$), and each row and column represent various types of component codes defined by $code_r$ and $code_c$, respectively. After the first code is mapped to the strips, the second code is mapped to the corresponding sub stripes that do not contain parity. The component codes can be, for example, a single parity code (SPC), a STAR code [35], an EVENODD code [6], and an X-code [8]. These codes can also be extended from two-dimensional to m-dimensional, denoted as GRID($code_1, code_2, ..., code_m$). When designed carefully, these codes can provide high fault tolerance and achieve good storage efficiency, while is the major trade off for erasure codes of storage systems. To assign all elements into an array, GRID codes provide two layouts: (1) a row-first layout and (2) a column-first layout, as shown in Fig. 3.1 which the matched codes are SPC code and EVENODD code.

Compared to other coding techniques, the authors claim that GRID codes require simpler operations and have better performance in large systems. Nevertheless, these codes are not MDS, and they require quite some overhead and lose approximately 20% storage efficiency compared to an optimal code.

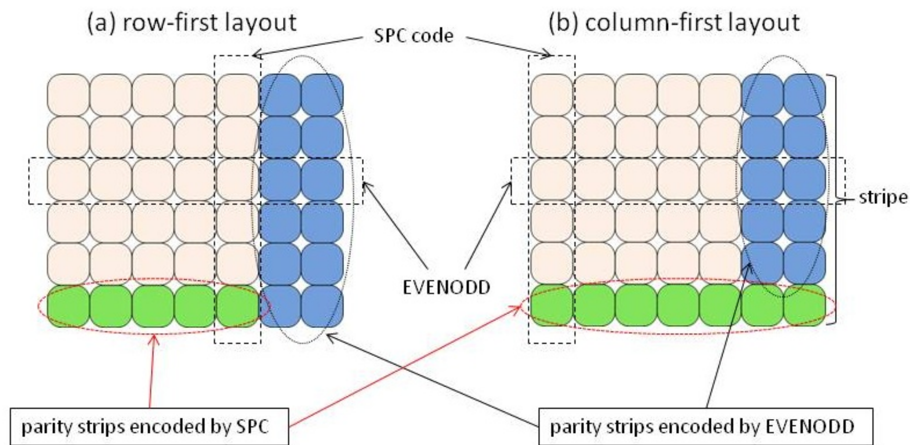


Figure 3.1: Two types of stripe layouts of GRID(SPC,EVENODD) codes

Motivated by these two classes of codes, GRID codes and regenerating codes, we explore efficient ways for large storage systems. The proposed idea is to generate layered protection, such as local protection, regional protection and global protection, to increase the fault tolerance, reduce the network traffic, and provide the scalability much needed for large systems, while still maintaining the benefits given by traditional erasure codes. We start by constructing a base graph (the first layer) to generate the base code with local parities (or local protection), and then move to the second layer called regional protection, which provides regional parity to connect local protection. Finally, the peel layer is defined as a global protection which can protect all the disks in the system by global parities.

In this chapter, we study a concatenation techniques to construct erasure codes for multi-layer hierarchical protection of data storages with the consecutive MDS and LT codes. We introduce the L groups of MDS codes which contain both information disks and parity disks called *local parity disks*. Then, we construct the second layer protection (or *global parity disks*) by randomly select degree d distribution mentioned in [25] and [26]. This technique will get both local and global parity disks connected and can protect all information disks. Our contributions from this work is that this code outperforms and can recover an arbitrary number of disk failures when the probability of disk errors is high.

3.1.1 Background of Luby Transform (LT) Codes

Luby Transform (LT) codes were the first practical rateless codes, where the number of encoding symbols that could be generated from the data was limitless [25]. LT codes have simple encoding and decoding procedures, and regardless of the erasure model, encoding symbols can be generated as needed

and sent over the channel until a sufficient number of symbols arrive at the decoder to recover the data. To create an encoded symbol, a set of d data bits are chosen to be XORed randomly and independently, where d follows the *robust soliton distribution.* For the decoding part, it uses the belief propagation algorithm [25] which the decoder will begin with the identifying encoded symbols of degree 1 thus yields the value of some other input symbols. For example, let x be the known input symbol, and then given $x \oplus y$, one can deduce y and so on.

Definition 3.1.1. [The Robust Soliton Distribution] [25] Let

$$\rho(i) = \begin{cases} 1/k & : \quad i = 1 \\ 1/i(i-1) & : \quad i = 2, \ldots, k \end{cases} \tag{3.1}$$

$$\tau(i) = \begin{cases} R/(ik) & : \quad i = 1, \ldots, k/R - 1 \\ 2e^2(R \log \frac{R}{\delta})/k & : \quad i = k/R \\ 0 & : \quad i = k/R + 1, \ldots, k \end{cases} \tag{3.2}$$

$$R = c \ln k/\delta\sqrt{k} \tag{3.3}$$

$$\beta = \sum_{i=1}^{k} (\rho(i) + \tau(i)) \tag{3.4}$$

Thus, the *Robust Soliton Distribution* $\mu(i)$ for $i=1,\ldots,k$ is

$$\mu(i)=(\rho(i)+\tau(i))/\beta. \tag{3.5}$$

The decoder receives K output symbols, and tries to recover the k input symbols from the neighbors of the output symbols in the bipartite graph. As shown in Fig.3.2, the decoding process will initiate all message symbols are unrecovered. Then, at the first step all degree one encoding symbols will get released to cover their neighbors. At any time t, these sets of recovered message symbols now form a ripple that later is selected randomly and processed. An encoding symbol which has degree one is removed and its neighbors are recovered (selected from the cloud contained the set of output symbols of reduced degree > 1) and added in the ripple. The process ends when the ripple is empty and all message symbols are recovered.

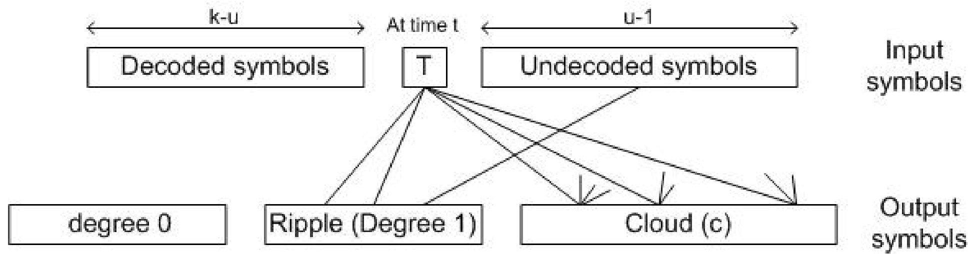


Figure 3.2: The decoding process when there are $u-1$ input symbols are undecoded

As shown in [25], LT codes require only $k+o(k)$ encoding symbols to decode with high probability, when using encoding symbols with average degree $O(\ln k)$. For affordable complexity, the constant average degree is needed and the decoding time will be $O(k)$. However, this cannot be done in LT codes. One can easily show that for every message node to have at least one neighbor when there are $O(k)$ encoding symbols, the average degree must be at least $\Omega(\ln k)$.

One of the disadvantages of LT codes is that they are not systematic. Thus, it cannot guarantee that the input symbols can be reproduced among a selected set of output symbols [22].

3.2 The MDS-LT Nested Codes

Borrowing ideas from systematically-layered graphs, we construct a multilayer MDS codes, namely *nested codes with hierarchical protection for distributed storage networks*. The local code may be either the CGR code proposed in the previous chapter, or other short-length, simple, and space-efficient erasure codes that have well-defined regular structures. The upper layer codes will subsume a random structure to allow for scalability.

1. Constructing the base code to provide local protection.
2. Constructing a second layer code to group several base codes together to generate common regional parities.
3. If necessary, constructing the third layer code to provide overall global parities that allow all the storage nodes and disks in the entire system to be connected and protected.

The basic structure of this idea is illustrated in Fig. 3.3. The primary benefit of such as structure is complexity, which comes in several aspects. First, to construct a good, simple and long erasure code for a large system all at once is extremely difficult. The layered structure offers an alternative approach by decomposing the difficult problem into several smaller and more tractable problems. In doing so, it also makes it possible to leverage the existing rich results in the coding literature. Second, even though a good long

code may be constructed, the implied communication overhead may be very large. This is because disks that are physically far apart from each other will likely participate in the same check, and hence to recover any failed disk will almost always involve the reading from several distant disks, causing a large input/output (I/O) overhead. Third, to deploy such a long code is extremely difficult. It may take several hours or several days to initially structure and encode all the data disks according to the long code, and the process should in general not be interrupted; that is, while implementing the erasure code in the system, new data, for example, should not enter into the system. Forth, after the initial implementation, scalability may also be an issue. As new data enter or two or multiple data centers merge, to efficiently and sufficiently protect the incoming data disks may be complicated. When the long code has a random structure, it is possible to add additional checks, but adding new checks may change the degree profile that was previously optimized and make it less efficient. Also the new data disks will likely have weaker protection than the old ones.

In comparison, the layered structure allows the data disks are firstly and foremost protected by neighboring disks. Only when local protection is insufficient (which should happen rather infrequently), will one resort to the regional protection and eventually global protection. As new data enter, new local clusters with local codes can be formed, and regional and global checks can be added on later and gradually. Below we discuss each of the three steps toward constructing the overall layered code.

3.2.1 The Code Construction

We detail the process to construct the hierarchical MDS-LT nested erasure code as followings:

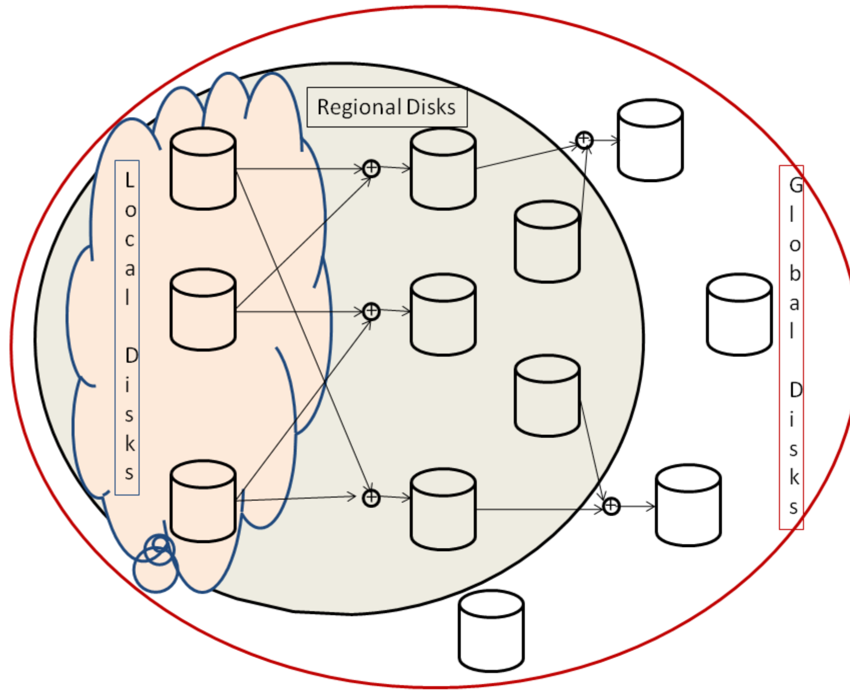


Figure 3.3: The basic structure of nested codes with Hierarchical protection for distributed storage networks

Step1: Base Code Construction

The CGR codes presented in the previous chapter, or other good erasure codes available in the literature, can be readily applied as a base code, as shown in Fig. 3.3. Local codes are generally short codes, and it is desirable to use MDS codes, especially codes with symmetric or cyclic structures to provide simplicity and space-efficiency. It is possible to use not just one specification of (CGR) MDS codes, but (slightly) different specification of (CGR) MDS codes in one system, to handle the quality difference of different types of hard disks that may co-exist in a storage system. For example, industry-grade hard disks are more robust and less prone to errors and disk failures, so a lower-protection erasure code with a higher code rate (i.e. payload) such as a $(7, 5)$

MDS code may suffice for clusters of these disks; on the other hand, near-line hard disks are more prone to failures, and hence a higher-redundancy lower-rate erasure code, such as an $(7, 4)$ MDS code or $(5, 3)$ MDS code, should be used in exchange for a stronger erasure protection capability.

Step2: The Second Layer Code Construction

In this second layer, several local codes will be grouped together to form regional clusters, upon which regional erasure codes are developed. All the storage disks of the local codes of the same regional cluster, be it data disks and parity disks, will all be treated as the systematic disks for the region code, and a common set of spare disks will be used as regional parities and coded across these local codes. Unless local codes, which are usually dense-graph codes, regional codes should in general be sparse-graph codes, so that they can minimize the communication overhead involved in each recovery.

Step3: The Third Layer Code Construction

The global code unifies all the regional codes in a way very similar to a regional code unifies the local codes, except that the scale here is even larger and the underlying graph is even more sparse.

As an experiment, we consider two coding layers. Our storage system is constructed from L disk groups. Among all L disk groups, there are kL information disks and $(n - k)L$ local parity disks which are constructed by any local (n, k) MDS code. The local code may be either the CGR code proposed in [24], or other short-length, simple, and space-efficient MDS codes that have well-defined regular structures. Then, we construct the second layer of data loss protection, namely "global parity disks" by using the idea of LT codes. To achieve the practical purpose, we generate the degree by the robust

degree distribution technique.

The system model we used in this work to construct the hierarchy nested erasure codes as shown in following assumptions [1].

1. We use and construct $CGR(K_2, C_5)$ [24] code which has the same performance and ability to recover disk failures as an $MDS(5,3)$ code for all L groups.

2. Construct a set of global parity disks by XORing data disks from totally L local MDS code groups follows by the rule of degree distribution of LT codes (which is in this work, we construct d-degree to random select which disks we will use for constructing global parity disks follow the method called "Robust Soliton Distribution"), so that the number of XORed data disks which is used to construct each parity will be different and depended on the probability of disk failures. From the randomly degree d we are chosen, we construct M global parity disks to be the second layer protection of this code, where the number of global parity disks (M) is computed by the appropriate ratio of M over L which $M = qL$, where $0 < q < 1$.

3. Assign all disks in the array as shown in Fig.3.4.

[1]Throughout this work proposed in here, we have generated only two layers which are the local and global layers, and they are encoded by MDS codes and LT codes, respectively. More than two layers construction will be left for the future works

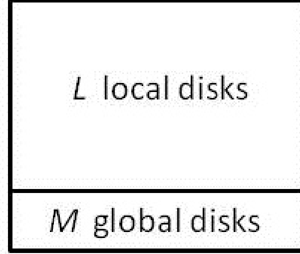


Figure 3.4: Code array structure where M global disks are all parity disks constructed from LT codes

3.2.2 The Consideration of Hierarchical Nested Erasure Codes

Since the hierarchy nested erasure code is the concatenated code from LT codes and MDS codes, we have to consider its characteristics from both codes.

Lemma 3.2.1. [25] For LT codes with Robust Soliton distribution, k original source blocks can be recovered from any $k + O(\sqrt{k}\ln^2(k/\delta))$ encoded output blocks with probability $1 - \delta$. Both encoding and decoding complexity is $O(k\ln(k/\delta))$.

Lemma 3.2.2. For any (n, k) MDS code with the minimum distance, $d = n - k + 1$. An MDS code can recover disk/node failures up to $t = n - k$ disks/node.

Simulation Results

In this section we illustrate all the simulation results of this code and the comparison with GRID codes. In this simulation, we assume that the (L, M, n, k)

codes are based on L groups of an (n, k) MDS code, and $M = qL$, where $0 < q < 1$. The rate of this code is given by $R = \dfrac{kL}{(L+M)n}$.

Fig. 3.5 illustrates the probability of residual disk errors after our hierarchy nested codes. For the number of local disks (L) are 150, 200, 250, and 300, and each disk is encoded by an $(5, 3)$ MDS code, we can see that their probabilities of disk errors are approximately the same. Since in this work all $(L, 0.1L, 5, 3)-$ nested erasure code have the same code rate, it is intuitively considered that their performance on any number of disks will be similar. At the lower rate of probability of disk failures ($P_e < 0.3$), our nested erasure code can make a 100% error recovery, and then for $P_e > 0.3$ we can recover and correct disk failures approximately by 20-30%. Note that in practical systems, the raw disk failure rate is very small ($P_e \ll 0.3$).

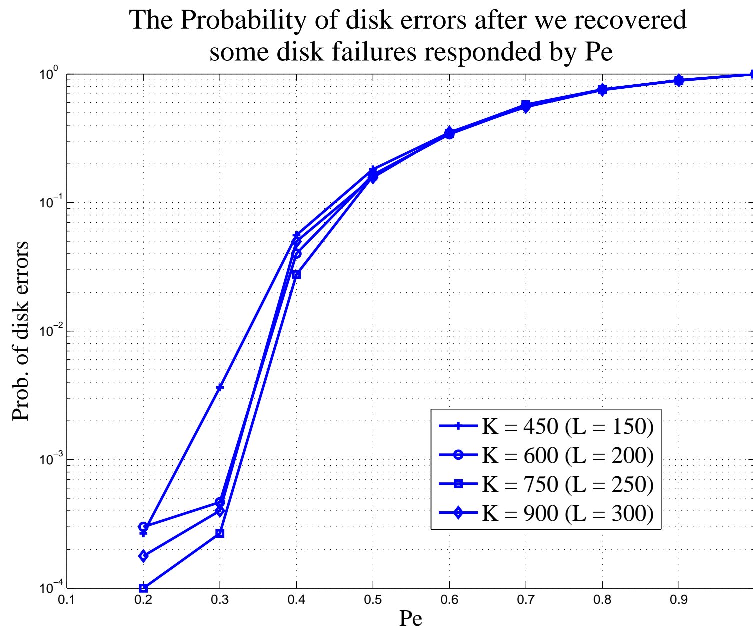


Figure 3.5: The probability of residual disk errors versus the raw disk failure rate (P_e).

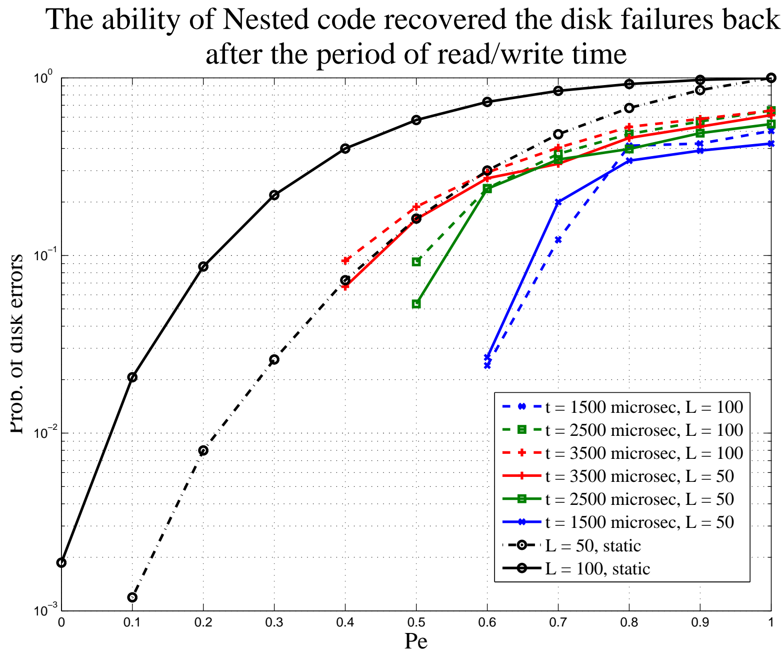


Figure 3.6: The ability of the hierarchy nested erasure code to recover failed disks in time period t.

Fig. 3.6 shows the dynamic simulation of MDS-LT nested codes based on $(5, 3)$ MDS codes in which the processing time for read/write operations and error recovery are taken into account. When disk errors occur, the failures will be detected and corrected before information can be read/written again. In this simulation, based on a real situation of reading/writing data on disks that whenever errors occur, the hierarchy nested erasure code tries to recover and correct failures as fast as possible. The result in Fig 3.6 shows the probability of disk failures under the condition of constant repair. The period of processing time (t) is set for recovering/correcting some failures before they reach some value of randomly errors as in the static case in Fig. 3.5. Thus, we can see that the probability of disk failures is greatly reduced.

Compared to Grid(STAR, STAR) codes in [39], as shown in Fig. 3.7, our

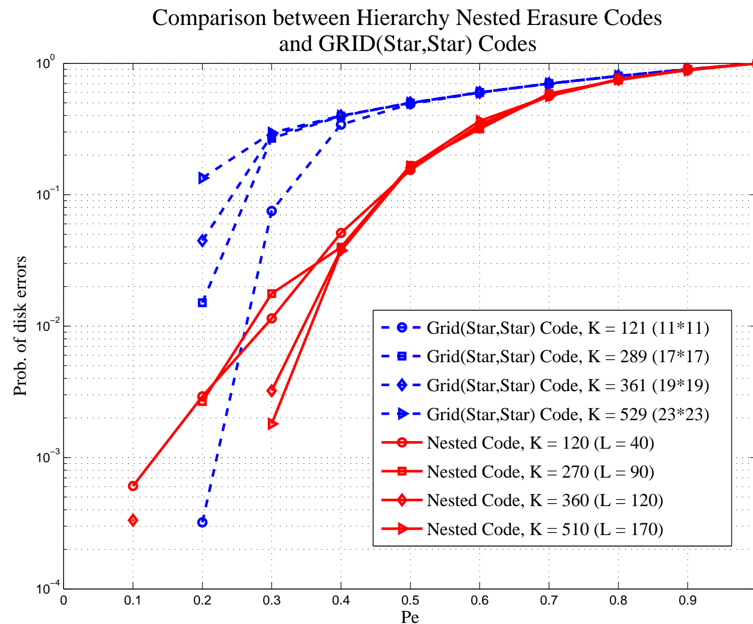


Figure 3.7: Comparisons the probability of disk errors between Grid codes and hierarchy nested erasure codes

code outperforms the Grid(STAR, STAR) at the similar number of information disks (K). We assume Grid(STAR,STAR) code can recover up to 15 failed disks. Fig. 3.7 shows that the GRID(STAR,STAR) code can handle the fault tolerance very well when the probability of disk errors (P_e) is low (not greater than 0.3), but its limitation is that the maximum number of disk correcting/recovery is only 15 failed disks ($t \leq 15$) so it will fail when P_e gets larger ($P_e \geq 0.4$). Nevertheless, the hierarchy nested erasure code performs well and has the ability to recover failed disks among thousands of disks in the network.

Analysis and Discussion

According to the characteristic of MDS codes, in each group it can tolerate up to $(n-k)$ failed disks. In order to reduce the complexity of decoding, we can recover the lost data back from their priority bits inside their own group without considering the others, and if there are too few parity bits left for decoding (there are more than $n-k$ disk failures), we can consider and use others in the global level selected by their relations. So, the local parity bits in each local disk have higher priority than all global parity disks and are the first to consider when disks are failed. To achieve the best performance, the value of parameters c and δ will be carefully chosen.

Consider the probability of error occurred from the finite length LT codes,

Theorem 3.2.3. [28] For any original code with k input symbols and $n = k(1+\delta)$ output symbols (encoded by LT codes) received for decoding with $\Omega_i = 1, \cdots, D$, where D is the maximum degree of an output symbol, the probability that an input symbol is of degree i when undecoded input symbols, $u = k+1, k, \cdots, 1$. P_u is a recursive form of the error probability when the decoding process fails.

$$P_{u-1}(x,y) = \frac{P_u\left(x(1-p_u)+yp_u, \frac{1}{u}+y(1-\frac{1}{u})\right)}{y} - \frac{P_u\left(x(1-p_u), \frac{1}{u}\right)}{y}, \tag{3.6}$$

where x, y are the input and output symbols, respectively, and

$$P_u(x, y) = \sum_{c \geq 0, r \geq 1} p_{c,r,u} x^c y^{r-1} \tag{3.7}$$

is the state generating function of the LT decoder when the cloud size is d, the ripple size is r, the associate probability in this state is $p_{c,r,u}$, and p_u is also shown in [28].

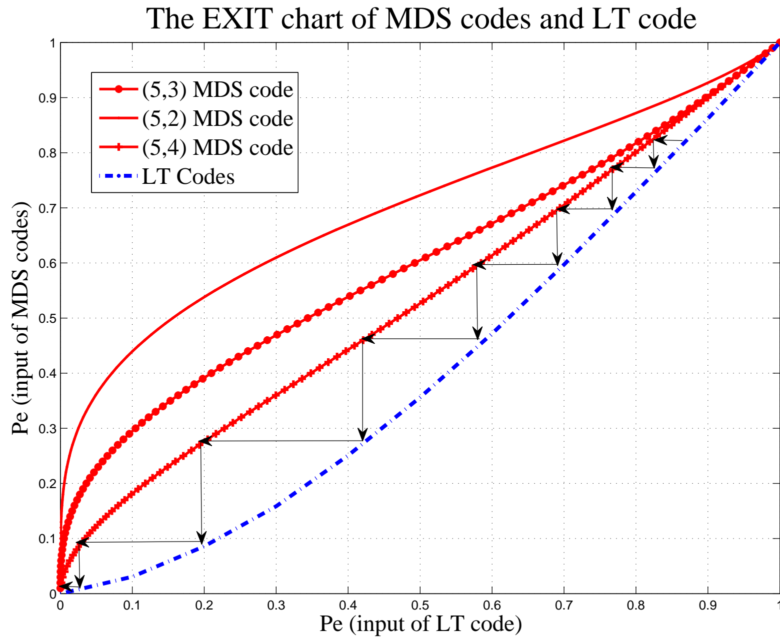


Figure 3.8: The EXIT chart of LT codes and MDS codes

We can also view the nested coding scheme as a concatenation, and use the extrinsic information transfer (EXIT) charts to visualize the performance and convergence.

From this EXIT Chart, we consider the error probability starting from the

high-error level. As shown in Fig. 3.8 the iterative decoding will asymptotically converge and eventually decode all erasures.

There are several trade-off issues in the scheme we developed here. The cumulative protection capability of all the layers provide the overall fault tolerance capability of the system, but how should the protection capability be divided and distributed among the three layers. Let α be the probability that a local code fails to recover a broken data disk, let β be the probability that the combination of the local codes and the regional code fails to recover a data disk, and let γ be the probability that all the three layers fail to recover a data disk. Following the high industry standard of five 9's, we may set γ be 0.99999. What are good values for α and β? Note that the bandwidth overhead is a function of α and β, as well as the size of local and regional clusters and the overall storage network. Hence, the problem may be formulated as a minimization problem (minimizing the bandwidth overhead) while meeting certain constraints for α, β and γ.

It should be noted that, depending on the actual size and other requirements and constraints of the storage networks, the system need not be limited to three layers. Shorter systems are better off using only two layers (i.e. local and global), while very large systems may go with four layers (or even more). In our initial study, we have experimented on a simple system with two layers. We specifically choose short MDS codes in the lower layer and use long LT codes in the upper layer. Several combinations of the code configuration are simulated, and it is found that the robust soliton degree distribution of the LT codes work better in the layered structure than the ideal soliton distribution. It is desirable to provide useful engineering rules for the number of layers and the preferred degree distribution for the upper layer codes.

In conclusion, this code is constructed based on the optimal MDS code and

then we construct the upper level (global) of parity disks by randomly selecting "d: degree" local disks and XORing them. This code not only protects the arbitrary disk failures, but also recovers the loss data back. Compared to the existing erasure codes such as GRID codes [39], the hierarchy nested erasure codes have a better performance when the disk error rate is more than 0.65 and more suitable to the distributed data storage system with more than hundreds of disks. However, this code does not efficient enough since its randomly degree selective technique might degrade the performance and cause too many overheads in the system from the limitless of LT codes resulted in unfair comparison with GRID codes. Thus, in general this code will not as efficient as the number of disks is increased in the systems. From these drawbacks, we will propose an idea to construct an erasure code with fixed structure and maintain the MDS property or nearly-MDS with low overhead.

3.3 The Horizontal-Vertical Single Parity Check (HVSPC) Codes

A major drawback of the previous MDS-LT nested code is that the random structure of LT codes does not promise definite or guaranteed recovery. Here we propose a fixed layered structure instead of the random structure in order to provide guaranteed performance.

For this code construction, we use various MDS codes as local codes for the local protection and use the horizontal single parity check (HSPC) code for the second protection, followed by a set of vertical single parity check (VSPC) codes. The code structure is shown in Fig. 3.9. The HSPC is parity set based on the horizontal SPC code which is computed by XORing all data in each row. The VHPC is parity set based on the vertical SPC code which is

computed by XORing all data in each column. The CC is check on check. Assume that the array size of MDS local array code is $x \times y$. The array size of the overall code is $(x+1) \times (y+1)$.

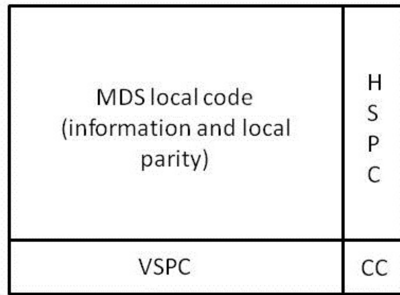


Figure 3.9: The array structure

However, this code has an special structure for MDS local codes. In stead of assigning each MDS code in each row or column, we assign them in a diagonal and cyclical fashion. As shown in Fig. 3.10, all the symbols are labeled by the same letter form a local MDS code.

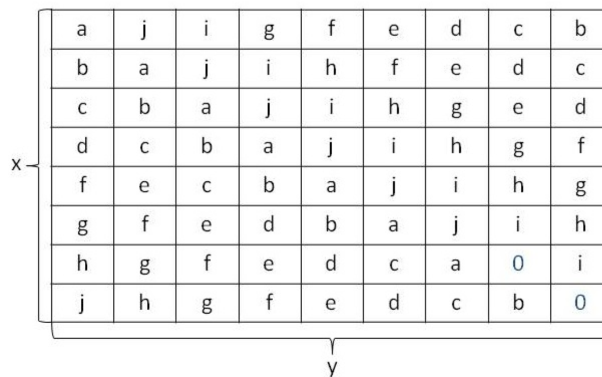


Figure 3.10: The organization of MDS local code in the array of size $x \times y$

A big advantage of this code is flexibility. We can construct the MDS local protection codes in any array size. In Fig. 3.10, we use (7,5)MDS codes and construct a parity array code with $x = 8$ rows and $y = 9$ columns. Thus, the overall code size is 9×10. Note that we will fill 0's for some empty blocks, as 7 is not divisible by $8 \cdot 9$.

3.3.1 Simulation Results and Analysis

The simulation results are shown in Fig.3.11 and Fig. 3.12. We construct this code based on various (n, k)MDS codes, including $(6, 2), (7, 5), (8, 3)$ and $(11, 9)$, and they can tolerate different numbers of disk failures.

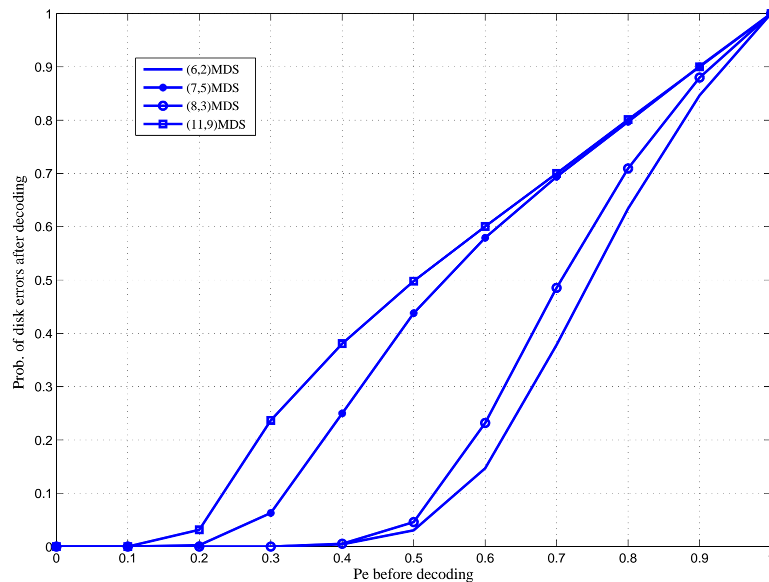


Figure 3.11: The probability of disk failures after applying the layered coding scheme.

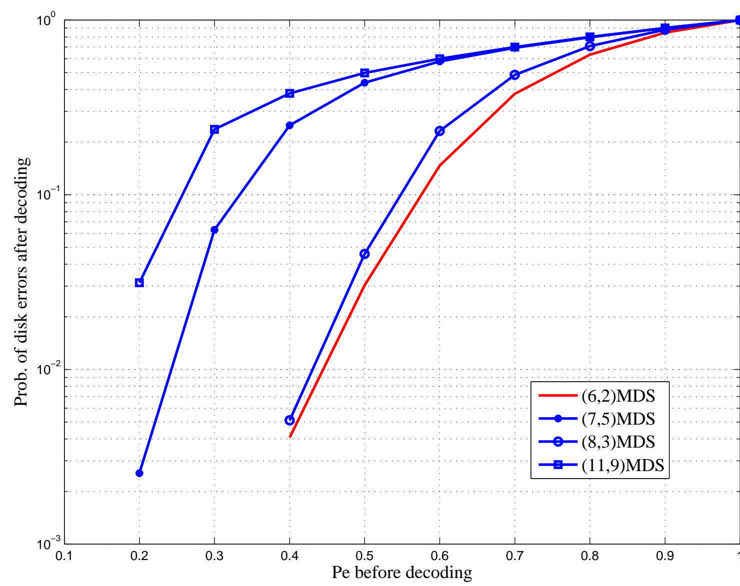


Figure 3.12: The probability (in log-scale) of disk failures after applying the layered coding scheme.

Both graphs show that the probability of disk failures is reduced after we apply this code to recover failures. However, there are still some unrecoverable erasures left. From the simulation, this code can recover most failures when the probability of raw disk failures (P_e) is low ($P_e \leq 0.2$). It is intuitive that the code based on MDS code with a larger number of redundancy can tolerate more disk failures. As expected, the codes based on $(6, 2)$ (4 overheads) and $(8, 3)$ (5 overheads) local MDS codes give a better performance in erasure tolerance than the ones based on less overheads.

The merits of this code are (1) scalability: we can construct this code in arbitrary any array size, (2) flexibility: we can use any size of MDS array codes as local codes, in accordance with the number of disk failures we would like to tolerate, and (3) fault tolerance: this code can handle a large number of erasures depending on how large the data center is and how many erasures the local MDS code can recover.

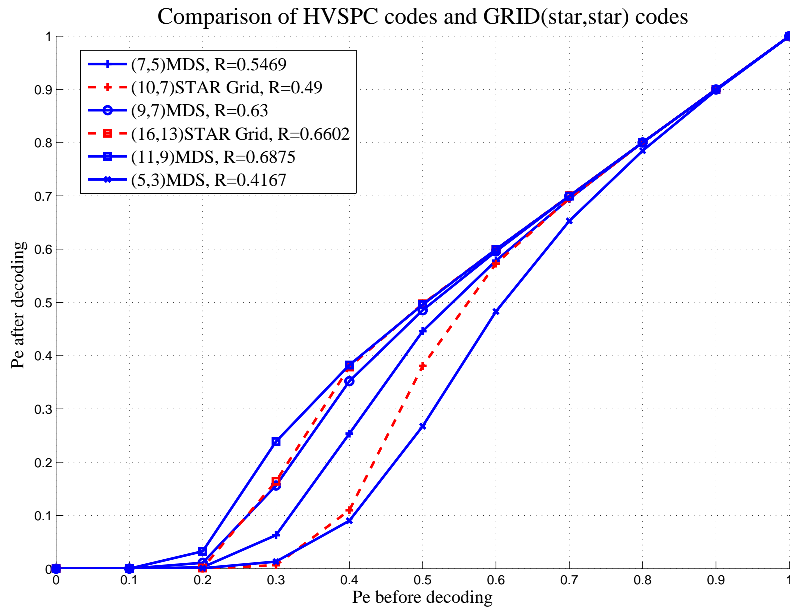


Figure 3.13: The comparison of HVSPC codes and GRID(STAR,STAR) codes

The results shown in Fig.3.13 are the probability of disk failures after we compare the performance of this HVSPC code with the GRID(STAR,STAR) code. For a fair comparison, we plot and show simulation results for both codes in the same code rates. This codes appear to perform on par with each other.

Like the GRID codes, HVSPC codes are not MDS codes so they do not achieve the optimal space efficiency, but our codes are more flexible to construct. We can apply any set of (n, k) MDS codes for the local protection and then cover them with the HVSPC codes for the global protection. This technique outperforms the one of the MDS-LT nested erasure codes by reducing the number of overheads and increasing the code rate.

3.4 Summary

This Chapter purpose is to construct the MDS or nearly-MDS codes for the distributed storage systems. We have proposed the concept of hierarchical structure where all disks are set by different priorities: the local, regional and global protection layers. In this work, we have introduced two layers which are local and global ones. The local disk arrays are protected and encoded by a set of MDS codes, then they are covered up by the global protection code. Both random and fixed code constructions have been analyzed and simulated compared to the GRID(STAR,STAR) codes. Unfortunately, our codes, MDS-LT nested codes and HVSPC codes, can handle less disk failures than GRID(STAR,STAR) comparison at the same code rate.

Additionally, there are some open problems for the future works. First of all, we will analyze its performance in terms of encoding/decoding complexity, and compute the maximum number of disk failures that HVSPC codes can

tolerate. Then, other classes of MDS codes may be applied instead of SPC codes in here in order to improve the ability to handle more disk failures and still maintain the property of MDS codes.

Chapter 4

Coding for flash memory

In Chapter 2 and Chapter 3, we have proposed ideas and strategies for code construction for disk array systems. In this Chapter, we will discuss and design coding techniques for flash memories which work differently from disk array systems. Since flash memories are free from any mechanical moving parts and consume less power, and since the price of the flash memories has dropped considerably (thanks to the more mature technology), it is believed that the trend to use flash memories in large-scale storage system would be in the (near) future. Coding for flash memories, together with other technologies, is a pillar supporting this new product.

4.1 Introduction

Flash memories are becoming an important part in many electronic devices such as MP3 player, PDAs, digital camera, or computer laptop due to its small size with large memory capacity. Unlike hard disk drives, flash drives do not

contain any internal moving part which cause to the mechanical failure issues. This reason makes flash drives are smaller and durable. However, there are two main limitations in storing and updating (read/write) data into flash memory. First, bits can only be cleared by erasing a large block of flash memory when it reaches the highest state level. Second, each block has a limit number in erasing process, after that it can no longer store or write any coming data.

In this research area, we are interested in multi-level flash memories rather than single-level flash memories, since the former has a higher storage density and a higher speed programming, which allows the number of stored bits in a cell to be drastically increased. However, one major drawback of flash memories is that although it can be read or programmed a byte or a word at a time in a random access fashion, it must be erased a block at a time. The theories on data representation for flash memories are introduced and discussed for a better improvement and development.

4.1.1 How Flash Memories work?

Since flash memories are non-volatile memories (NVM), they work differently from the traditional disks. They do not contain any moving/mechanical part which may cause some noise like in magnetic disk drives, therefore being more robust. They are also known as solid-state storage device because of none of any moving parts – everything is electronic instead of mechanical.

NVM memories have been continuously growing in the industrial market in the past few years and further growth in the near future, especially for flash memory. Flash memory is a forms of chip called EEPROM or Electronically Erasable Programmable Read Only Memory which contains a grid of columns and rows within a cell that has 2 transistors known as a floating gate and a control gate at each intersection as shown in Fig.4.1. They are separated by

a thin oxide layer. The floating gate links to the row or word line through a control gate. In order to update/change a cell state to be '0', the *Fowler-Nordheim tunnelling* is required, otherwise the cell state has a value '1' (as long as the link is in place) [2]. On one hand, if we use two discrete charge levels to store data, the cell is called *"single-level"* cell (SLC) and can store one bit. On the other hand, if we use more than 2 ($q > 2$) discrete charge levels to store data, the cell is called *"multi-level"* cell (MLC) and can store $\log_2 q$ bits [63].

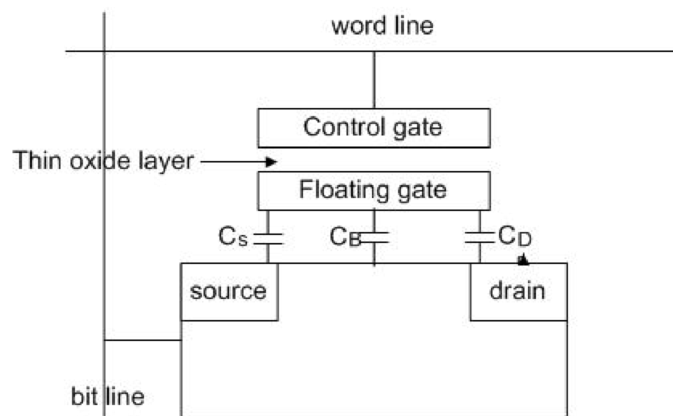


Figure 4.1: Schematic cross section of flash memory.

The tunnelling mechanism is the process to alter the placement of electrons in the floating gate. The charge (e.g.,electrons) comes from the bit line to the floating gate, and drain to the ground. The excited electrons are pushed through and trapped on other side of the thin oxide layer, giving it a negative charge. These negatively charged electrons act as a barrier between the control gate and the floating gate. The charge is stored in the floating gate layer of the transistor. If the flow through the gate is above the 50 percent threshold, it has a value of 1. When the charge passing through drops below the 50-percent threshold, the value changes to 0. A blank EEPROM has all of the gates fully open, giving each cell a value of 1. The electrons in the cells of a flash-memory chip can be returned to normal ("1") by the application of an

electric field, a higher-voltage charge. To inject charge into the cell is called "writing/programming," remove charge is called "erasing," and measuring the charge level/state is called "reading". Flash memory uses in-circuit wiring to apply the electric field either to the entire chip or to predetermined sections known as blocks. This erases the targeted area of the chip, which can then be rewritten. Flash memory works much faster than traditional EEPROMs because instead of erasing one byte at a time, it erases a block or the entire chip, and then rewrites it [63].

Flash memory has asymmetric properties since the changes in the cell levels have asymmetric distribution where they are frequently changed in the up and down direction, and the errors in different cells can be correlated. Due to this property, it is easy to increase a charge level, but very expensive to decrease it because to lower a cell's level, block erasure is needed. From this problem, the coding scheme for rewriting data is interested and would lead to allow data to be rewritten many time before block erasure is needed and hence can be lengthen the lifespan of flash memory.

4.1.2 Literature Reviews

Flashed back to the single cell flash memory which was studied in the write-once memories (WOM) codes in"How to reuse a *write-once memory*" [47], the write-asymmetric memory (WAM) could be used several times. A 'write-once' bit position defined as a 'wit' contained two states in each cell (for example, punch cards), then they came up with the lemma that only 3 wits were needed to write 2 bits twice without resetting any cell. Since then, there were numerous papers that were motivated and built based on their work such as [48]-[50], to name but a few. In the WOM model, its codes needed to solve the problem that what the minimum number of cell n required to stored k bits

t times should be. So, the WOM codes was designed with the WOM-rate R^t with the t writes was defined in [55] as the ratio of the total number of bits written to the memory, kt, to the number of cells, n, $R^t = \frac{kt}{n}$. In [48], they studied the generalization WOM and reused them for successive cycles under the condition that the encoder knows the previous state of the memory, but the decoder does not. This work can be the extension of Wolf, Wyner, Ziv, and Korner for the binary WOM to the generalized WOM. The cost to rewritten the data in WOM also presented in [50]. They gave a characterization of the basic quantity of WOM and showed the related quantities that are useful for following works.

However, in recently the multi-level flash memory cell is invented in order to increase the number of stored bit in a cell. This property as well as its high storage density and high speed programming has made the flash memory popular for the portable devices and technologies. Flash memories have properties that work differently from traditional memories since they have many levels of cell state which are used to store data: (1) a block erasure when any cell is full, a block contained such cell must be erased, (2) the direction of updating: there are only two operations on cells which are increasing the charge level (charging) or erasing the contents of the cells (discharging), and (3) the limited lifetime of cells because the number of block erasures is finite.

Recently, in [45], Jiang discussed the generalization of error-correcting WOM codes for the flash memory model. Consider a block of multi-level flash memory with many levels of cell states for storing data. Assume that each cell has q states: $0, 1, ..., q-1$, where currently q typical ranges from 2 to 4 states, but the possibilities of q are in a much wider range between 2 and 256. Note that Flash memory offers random-access reading and programming operations but it cannot offer random-access rewrite or erase operations. The state of a flash memory can be easily increased from a lower state to a higher one by injecting an electron into the cell level, but to decrease the state of a

cell a flash memory is difficult and is typically achieved by erasing the whole block and re-programming (resetting) all the cells in the block [45]. In general, the block size can be thousands of or hundreds of thousands of cells, so to erase the whole block is not only time consuming, but it also degrades the efficiency and quality of flash memories. Since it is much more costly to decrease than to increase the state of cells, decreasing cell states should be avoided or delayed as much as possible.

Floating codes [45] are designed for k variables taking values $\{0, ..., l-1\}$ represented data stored in a block of n $q-$ary cells. This code consists of 2 functions: decoding function: $\{0, 1, ..., q-1^n\} \rightarrow \{0, 1, ..., l-1^k\}$, which maps a cell state vector to a variable vector, and update function: $\{0, 1, ..., q-1^n\} \times \{1, 2, ..., k\} \rightarrow \{0, 1, ..., q-1^n\}$, which updates the block to reflect a data change in the selected element of the variable vector. This code guarantees the number of rewriting times (t) as

$$t \leq \begin{cases} [n-k(l-1)+1](q-1)+\lfloor \dfrac{[k(l-1)-1](q-1)}{2} \rfloor & \text{if } n \geq k(l-1)-1 \\ \lfloor \dfrac{n(q-1)}{2} \rfloor & \text{if } n < k(l-1)-1 \end{cases}$$

The floating codes whose average block erasure period is better than the existing one in [45], [46] are proposed in [56]. The codes are based on the Gray code and have a simple implementation by concatenating the codes. Codes for $n = 2m$, $k = 2$, and $l = 2$ can be obtained where n is the number of cells in a block, k is the number of input information symbols, l is the number of levels of input, and m is a positive integer.

In addition, the *multidimensional flash code* [54] improves on the floating

codes in [45] to achieve more precise measure of optimality than *the asymptotically optimality.* The two main contributions of this work are a new measurement called *write deficiency* to decide how good a code is, and their new floating codes. The purpose of constructing this code are to eliminate the need for discrete cell level, and to overcome the *overshoot errors* (errors in which too many electron are added), which is a serious problem that reduces a writing speed during cell programming. The both basic and enhanced multidimensional construction constructed recursively on k and assume that q is only an odd number are successfully introduced.

Compared to floating code [45, 46], this code construction is simpler in the case of storing 2 bits using an arbitrary number of q-level cells. However, for the case of 4 bits, the drawback is it still has high write deficiency which is depended on the number of cells.

Furthermore, the *indexed code* [45] try to sacrifice some small number of cells as indexed cells to remember which cell group store which variable group by using the permutation of the number of group. This strategy is complicated and hard to implement when the number of cells, n, is really large and it has to sacrifice more cells to store the permutation. It also needs a mapping table between permutations and variable vectors.

Therefore, an important goal of this research area is to maximize the limited life cycle of a flash memory, or, in the other way, to maximize the number of times data can be rewritten between two block erasures [45]. The key questions thereof are: How should flash memory change its data structure or data representation and how should the cell states change as a bit representation(variables)? The solution lies in the design of good codes that map digital data to cell states and vice version. Unlike the erasure codes discussed in the previous two Chapters which are *error-correcting codes* or *channel codes*, here the codes are *data representation codes* or *source codes.*

Present research work on coding for flash memories investigates how to effectively/efficiently write, read, and program data, and then analyzes their performance. Another objective this research is that to correct some error of data representation. *Rank Modulation* codes [57],[58],[60] claim to be the first code that can correct errors/erasures written in flash memory. It is a scheme that uses the relative orders of cell levels to represent data. Instead of using the real value of data stored in cells, the ranks of cells are used and mapped to the information bits. The charge level in each cell induces a permutation that can measure the corruption of a stored information. This code makes more robust to program flash memory cells. Several works later investigate the potential error-correcting codes to improve the reliability of flash memories such as in [61]. This coding scheme is based on the premise of cells whose levels are higher than other need not to be increased, but this introduces errors called "controllable errors" in the recorded data and then can be corrected by this code. However, the complexity of the encoder and decoder is essentially involved in identifying the controllable errors, so in the practical implications the encoder/decoder implementation is more complex than the traditional flash coding schemes which aim to maximize the number of writes only.

In this dissertation, we study and focus on the multilevel flash memory where every cell has $q > 2$ states ($q \in 0, 1, 2, ..., q-1$). It will change the state of cell by injecting (programming) or removing (erasing) charge into/from the cell [62]. To avoid/delay the erasing process, we try to extend the life cycles of flash memories by maximizing the number of writes as much as possible. We will introduce the new performance measurement termed the "*word-writes*" to record the number of writes from the user's view before the block erasure is required. In addition, we consider and discuss two coding techniques to construct two new codes for different applications for flash memories which is the topic of this Chapter.

4.1.3 The Number of Writes Consideration

To consider the performance of flash memories, many researchers define some definitions and measurements to analyse their performance, especially in term of the number of writes they can achieve and guarantee. Unlike the traditional channel codes, coding techniques for flash memory cannot be measured their performance by the same method in terms of code rate R, minimum distance d_{min}, or encoding/decoding complexity.

The efficiency of a flash code may be measured by its best-case, average-case, or, more commonly, worst-case (i.e. guaranteed) write efficiency, which is limited by the most undesirable sequence of variable vector updates that leads to the least number of state vector updates (valid programming) before any cell exceeds its maximum state level. Formally, we have the following definitions:

Definition 4.1.1. A flash code *guarantees t bit-writes*, if every sequence of up to t bit-writes in the variable vector is possible, i.e. can find the corresponding sequence of update rules in the transition map before a block erase.

Definition 4.1.2. [54] Let $\mathbf{x} = (x_1, x_2, ..., x_n)$ denote the state vector of the n cells, where $0 \leq x_i \leq q - 1$ denotes the level of the ith cell. The *weight of the state vector*, or, simply, the *cell state weight* is defined as $w_x = \sum_{i=1}^{n} x_i$.

It should be noted that, all the research work in literature has, by default, considered a "write" operation as a "write of a single bit", and therefore has used the number of bit-writes, as a figure of merit. Further, under the assumption that each write operation will increase the cell state weight by at least one, a trivial upper bound for the number of bit-writes can be derived:

$$t \leq n(q-1) \tag{4.1}$$

This bound can be achieved by, for example, $k = 1$. With this upper bound, a concept of *write deficiency* results, which is defined as the difference between the guaranteed number of bit-writes of a flash code and the upper bound $n(q-1)$ [54]. The write deficiency is zero when a code is optimal. Additionally, a tighter upper-bound on t is also derived in [46]:

Theorem 4.1.3. [Bit-Write Upper-Bound] [46] For all $(n, k)_q$ flash codes that guarantee t bit-writes before erasing, t is upper-bounded by[1]

$$\begin{aligned} t &\leq T_b(n,k,q) \\ &\triangleq \begin{cases} (n-k+1)(q-1)+\lfloor \frac{(k-1)(q-1)}{2} \rfloor, & \text{if } n \geq k-1, \\ \lfloor \frac{n(q-1)}{2} \rfloor, & \text{if } n < k-1. \end{cases} \end{aligned} \tag{4.2}$$

4.2 The Word-write Efficient and Bit-write Efficient (WEBE) Codes

Motivated by the construction of flash code, we generate a new class of code called the "*Word-write Efficient and Bit-write Efficient*" code or *WEBE* code that not only consider a bit-write, but also a word-write which the definition will be provided below. This section considers the design of flash codes. Unlike all the previous work that targets optimal bit-write efficiency, here we

[1] Throughout this part, we assume the variable vectors are binary vectors, as in today's digital computer and communication systems. The general case of arbitrary variable alphabet size l can be found in [57] [58] [54].

emphasize word-write efficiency. It may appear that to write a word inevitably involves the update of individual bits, and hence it seems sufficient to focus on maximizing bit-writes.

Definition 4.2.1. For every time some/all bits need to be written/updated, if there are always available cells to be increased their level, it can be counted as one *word-write.* Otherwise, either errors occur or this block needs to be erased.

However, the fundamental difference is that, when word-writes are in concern, the intermediate steps of updating each individual bits need not (and probably should not) be explicitly expressed. That is, when dealing with a word-write that involves the update of several bits, instead of treating it as a sequence of individual bit-writes in succession and hence associating it with a cell state of a rather high "rank", it may be beneficial to provide a "shortcut" and assign it a separate cell state of low rank. This could significantly reduce the rank-increase in a worst-case scenario (when a word-write involves all the bit-writes) and therefore increase the supportable number of word-writes. We will show that bit-writes and word-writes are indeed related but different philosophies that will in general lead to rather different designs, and that optimality in one does not necessarily imply optimality in the other.

Specifically, we propose a class of word-write efficiency codes for $k=2$ and arbitrary n and q, and analyze their performance. Our codes are simple but guarantee more word-writes than the existing (bit-write) optimal codes [46] [54]. In addition, we provide a generalization of this code for arbitrary k, n and q, and also discuss its performance and future works.

4.2.1 Problem Formulation and New Concepts

We consider the design of representation codes for flash memories, or, simply, flash codes. A set of n q-level cells have altogether q^n possible state values. An $(n, k)_q$ *flash code*, as defined in [54], is a coding mechanism that arranges n q-level cells to store $k < log_2(q^n) = n \log_2 q$ variable bits, that is, storing less bits than possible, so that variable updates can be achieved through cell programming rather than trigger a block erasure. In practice, it is common to have $k \leq n$, but not necessarily since we may have a single 8-state cell that can be used to represent 2 variable bits. One can also define $\frac{k}{n} \log_2 q \in (0, 1)$ as the *code rate* for the flash code, but the code rate is not a very important concept in flash codes.

A conventional code, such as a channel code or a (lossless) source code, is completely defined by a codebook, or, a map between the source (variable vectors) to the codewords (state vectors). In comparison, a flash code generally involves two maps: the *decoding map* and a *transition map* [54]. A decoding map specifies the value of the corresponding variable vector associated with each (valid) cell state vector, and is similar in flavor to the conventional concept of codebook. The transition map, which is unique to flash codes, specifies the set of rules of updating/programming the state vector, such that each update corresponds to a possible change in the variable vector. Sometimes, it is convenient to represent the transition map using a directed graph, such as in float codes [46] [54]. Note that the state (the charge level) on a cell can only be increased or set to zero.

All the previous work has laid solid foundation in the theory and practice of flash codes. However, instead of focusing solely on the number of bit-writes, in this work we propose to also consider *word-write*, i.e. write of the entire variable vector, and to design flash codes that can maximize the guaranteed

number of word-writes. Word-writes are relevant and of paramount practical interests, because real-world applications such as digital computers usually perform the write operation in the *word* level, rather than the bit level, where a word can be, for example, 8 bits, 16 bits or 32 bits. Each word-write may consist of anywhere from 1 to k bit-writes, and hence bit-write efficiency does not linearly transform to word-write efficiency. We claim that the design for bit-writes and the design for word-writes are two related but different philosophies that will in general lead to different designs. Since a bit-write optimal code may not be equally optimal in word-writes and vice-versa, we propose to account for both criteria, and to design codes that are both Word-write Optimal and Bit-write Optimal (WOBO), or, at least Word-write Optimal and Bit-write Efficient (WOBE), or Word-write Efficient and Bit-write Optimal (WEBO).

Following the idea of WOBO, we first develop the following concepts and bounds:

Definition 4.2.2. A flash code *guarantees t word-writes*, if every sequence of up to t variable vector writes/updates are possible before triggering a block erasure.

Theorem 4.2.3. [Word-Write Upper-Bound] Suppose an $(n, k)_q$ flash codes guarantees t word-writes before erasing. t is upper-bounded by

$$t \leq \min\left(\left\lfloor \frac{q^n - 1}{2^k - 1} \right\rfloor, T_b(n, k, q)\right), \tag{4.3}$$

Proof. Since the set of the word-write sequences subsumes all the bit-write sequences, the number of guaranteed word-writes cannot exceed the number of guaranteed bit-writes $T_b(n, k, q)$. We now show $t \leq \lfloor \frac{(q^n - 1)}{(2^k - 1)} \rfloor$. There are altogether q^n different states for n q-level cells. Since the flash code must be unequivocally decodable, each state can represent at the most one variable

value. Let the k-bit variable vector start from the all-zero value. To guarantee one word-write would require all the other $(2^k - 1)$ variable values to have distinct representations from the cell states. Hence, it takes at least $1 + (2^k - 1)$ distinct states (including the initial state) to achieve one arbitrary word-write.

In the tth ($t \geq 2$) word-write, consider the variable value that corresponds to the state (or one of the states) having the largest state weight (among all the already-allocated states). Clearly, this variable may be updated to any of the other $(2^k - 1)$ possible values, and hence requires an additional set of $(2^k - 1)$ cell states to represent them. None of these $(2^k - 1)$ cell states may be re-cycled from any of the previously allocated states, because of the asymmetry in state reduction. Hence for t arbitrary word-write, it requires the flash memory to have at the least $1 + t(2^k - 1)$ distinct cell states, which results in $t \leq \lfloor \frac{(q^n - 1)}{(2^k - 1)} \rfloor$. □

Definition 4.2.4. If a flash code that guarantees t word-writes, then its *word-write deficiency* is defined to be $\delta_w = \lfloor \frac{(q^n - 1)}{(2^k - 1)} \rfloor - t$.

Remark 4.2.5. The bound in (4.3) is simple, and not tight in general. However, it is tight and achievable for certain non-trivial cases. Below we show an example of $(n, k)_q = (3, 2)_2$ flash code that achieves this bound with equality.

Example 1: Consider using 3 2-level cells to represent a variable word of 2 bits. From Theorem 2, the maximum number of guaranteed word-writes is 2. The flash code in Fig. 4.2 achieves the bound with equality and is therefore word-write optimal (WO). An acyclic directed graph is used to represent the transition map, with the decoding map also embedded. Each state is denoted by "state-value / variable-value."

Theorem 4.2.6. The relation between bit-writes and word-writes:

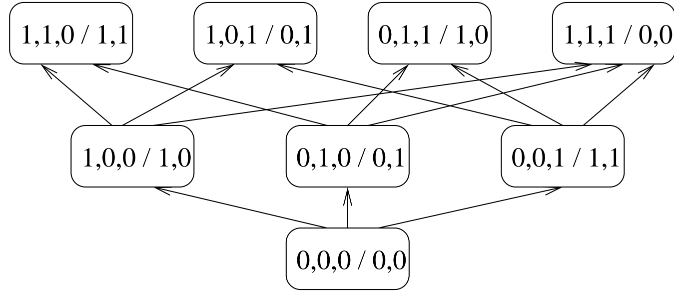


Figure 4.2: A $(3, 2)_2$ flash code that achieves the maximum word-write efficiency 2.

(i) A flash code that guarantees t word-writes also guarantees t bit-writes.

(ii) A flash code that guarantees t bit-writes does not necessarily guarantee t word-writes.

(iii) Bit-write optimality does not necessarily imply word-write optimality in terms of guaranteed writes.

Proof. The statement in (i) is easy to prove, as any sequence of t bit-writes is also a sequence of t word-writes. To show (ii) and (iii), it is enough to show a counter-example, Example 2 in Fig. 4.3.

Example 2: The $(3, 2)_2$ flash code in Fig. 4.3 is an instance of the bit-write optimal code (termed floating code) in [46]. It achieves the bit-write bound $T_b = 2$ in (4.2). However, this code only guarantees 1 word-write (e.g. a word-write sequence $(0, 0) \rightarrow (1, 1) \rightarrow (0, 1)$ cannot be satisfied.), and hence falls short from the maximum possible word-writes (which is also 2, see the example in Fig. 4.2).

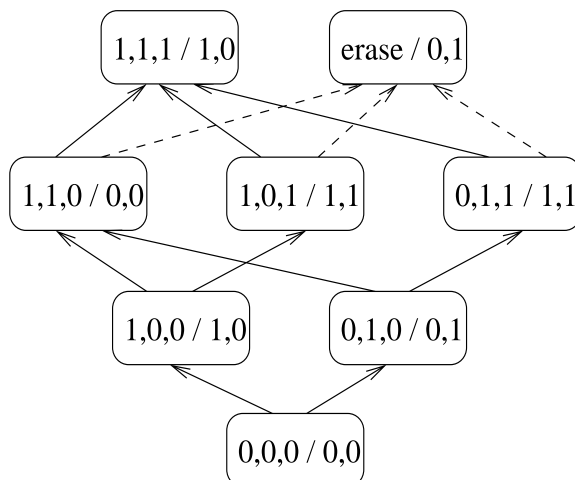


Figure 4.3: A $(3,2)_2$ flash code (floating code in [46]) that achieves the maximum bit-write efficiency, but not the maximum word-write efficiency.

It is clear from the examples in Fig. 4.2 and 4.3 that the design for bit-write efficiency, a practice that has prevailed the literature, does not guarantee word-write efficiency. The question then arises as whether word-write efficiency will automatically imply bit-write efficiency. Our answer is no. In general, we believe that the set of bit-write optimal codes and the set of word-write optimal codes relate to each other as shown in Fig 4.4, and the intersect is not empty, i.e. the code in Example 1 in Fig. 4.2 is an example of WOBO code. □

4.2.2 Design WEBE Codes for $k = 2$

In this case, we introduce the WEBE code that can be represented by two binary bits, whereas the general case for an arbitrary k is shown later in next section.

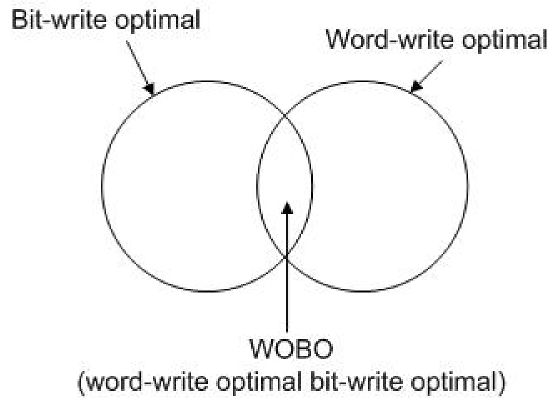


Figure 4.4: Relation between bit-write optimality and word-write optimality.

The Special Case of $n = 3$

Here, for the sake of simply illustration we first describe the code construction for there are three cells in a block and represented by two bits. It is clear from the previous discussion that one should account for both word-writes and bit-writes, and design WOBO codes that achieve both bounds. Note that WOBO codes exist, but may not for any parameters. In what follows, we will present a design of WEBE (word-write efficient and bit-write efficient) codes for $k = 2$ and arbitrary n and q. The special case of $n = 3$ and $q = 2$ results in the WOBO code in Example 1.

To help illustrate, we first discuss the code in terms of $n = 3$, and then generalize it to arbitrary n. We will present bounds on its write efficiency, and compute them with the existing bit-write optimal codes.

The proposed $(3, 2)_q$ code is based on a simple but profound observation: A 2-bit (binary) variable has only three possible word-updates, change the first bit, change the second bit, and change both bits, where each can be replaced by the combination of the other two. For example, if one wishes to change

only the first variable bit, he can either perform “change first”, or perform “change second” followed by “change both” (or the other way around since the order does not matter).

This motivates us to employ $n = 3$ cells to track and record these three kinds of word-updates respectively, and, if any cell becomes saturated, the other two can come to help, until, of course, a second cell also becomes saturated, in which case, an arbitrary word-update cannot be performed/recorded. A rigorous mathematical definition of the code is given in Algorithm 1.

Algorithm 1: A class of $(3,2)_q$ WEBE codes:

Notation:

$\mathbf{x}=(x_1,x_2,x_3)$: the state vector, where $x_i=0,1,...,q-1$.

$\mathbf{u}=(u_1,u_2)$: the variable vector, where $u_i=0$ or 1.

Starting state: $\mathbf{x}=(0,0,0)$, $\mathbf{u}=(0,0)$.

Decoding Map:

$$u_1 = mod(x_1, 2) \oplus mod(x_3, 2), \tag{4.4}$$

$$u_2 = mod(x_2, 2) \oplus mod(x_3, 2), \tag{4.5}$$

where $\oplus$ denotes binary addition (i.e. exclusive OR, XOR), and $mod(x_i, 2)$ is a modulo 2 operation.

Transition Map:

- $(u_1,u_2) \to (u_1+1,u_2)$:
  Increase x_1 by 1 if possible; otherwise, increase both x_2 and x_3 by 1.

- $(u_1,u_2) \to (u_1,u_2+1)$:
  Increase x_2 by 1 if possible; otherwise, increase both x_1 and x_3 by 1.

- $(u_1,u_2) \to (u_1+1,u_2+1)$:
  Increase x_3 by 1 if possible; otherwise, increase both x_1 and x_2 by 1.

The WEBE codes constructed in Algorithm 1 have the following properties. Each word-write causes the overall state weight to increase by either 1 or 2. For a cell state vector $\mathbf{x}$ with weight $w = \sum_i x_i$, we know it has gone through a minimum of $\lceil \frac{w}{2} \rceil$ word-writes, and a maximum of w word-writes. If the

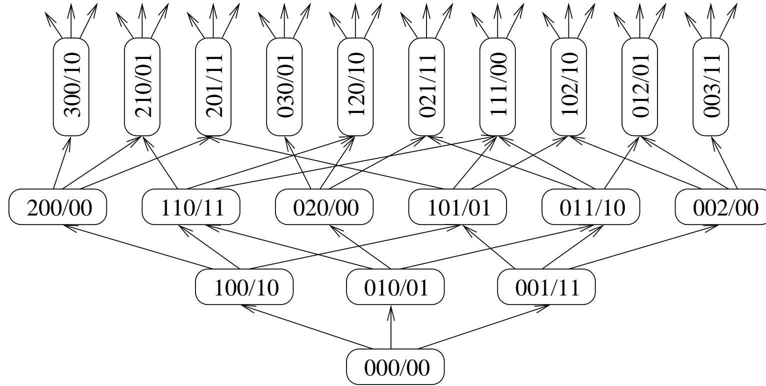
weight $w \leq (q-1)$, then the cells have gone through exactly w word-writes, and we know what these writes are, although we do not know the exact order at which they are performed. Further, the all-saturated state ($x_i = q-1, \forall i$) always corresponds to the all-zero variable $(0,0)$ irrespective of q.

Theorem 4.2.7. The $(3,2)_q$ code in Algorithm 1 guarantees $t=2(q-1)$ word-writes (i.e. worst case), and can support up to $n(q-1)=3(q-1)$ word-writes in the best case.

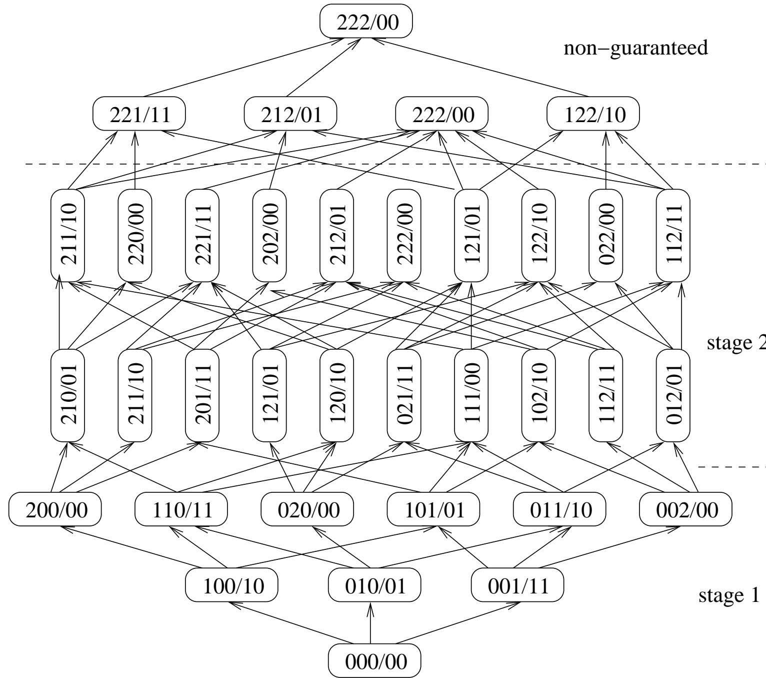
Proof. The best-case performance bound is trivial. We prove the worst-case by showing $t \leq 2(q-1)$ and $t \geq 2(q-1)$.

Every word-write increases the state weight by either 1 or 2. In any case, the first $(q-1)$ word-writes (call it Stage 1) will not cause any cell to saturate, and hence each word-write increases the state weight only by 1, resulting in a total state weight of $(q-1)$. Since the maximum state weight can be $n(q-1)=3(q-1)$, so there remains $2(q-1)s$ state weights for Stage 2. It is possible that Stage 1 has saturated a cell, such that every word-write in Stage 2 causes the state weight to increase by 2. Hence stage 2 can support at the most $(q-1)$ word-writes, that is a total of $t \leq 2(q-1)$ word-writes that can be supported.

We now show $t \geq 2(q-1)$. Consider an arbitrary state vector $\mathbf{x}$ that has supported b word-writes and can no longer support another arbitrary word-write. From the transition map, at least two of the cells are saturated. Without loss of generality, assume $\mathbf{x} = (q–1, q–1, a)$ $(0 \leq a \leq q–1)$. We show $b \geq 2(q–1)$ by contradiction. If $b \leq 2(q-1)-1$, then the cells must have gone through at least $(a+1)$ word-writes each of which has caused the state weight to increase by 2. (This is because the total weight is $2(q–1)+a$, and the first $(q–1)$ word-writes are always weight-1 word-writes. So the remainder $b-(q-1) \leq q-2$ word-writes must cause the weight to increase to $(q-1+a)$.) These $(a+1)$



(A) $(3,2)_q$ code with "Stage 1" (first $(q-1)$) word-writes.



(B) A $(3,2)_3$ WEBE code that guarantees $2(q-1)=4$ word-writes and supports up to $n(q-1)=6$ word-writes.

Figure 4.5: The proposed $(3,2)_q$ flash code.

weight-2 word-writes must happen after some cell is saturated, and must cause the other two cells to each increase by $(a+1)$. That is, none of the three cells can be in a level smaller than $(a+1)$, which contradicts with the supposed cell state $\mathbf{x} = (q-1, q-1, a)$. □

Example 3: Fig. 4.5 presents a graph illustration of the proposed $(3,2)_q$ code. Fig. 4.5(A) shows Stage 1 (self-sufficient stage) for arbitrary q, and Fig. 4.5(B) shows the complete diagram for $q = 3$, including Stage 1, Stage 2 (mutual-leveraging stage) and beyond (non-guaranteed word-writes). It is clear, from the proof of Theorem 4 and from this example, that it is a big benefit in the design for the cells to be able to leverage each other, as the mutual-leveraging stage supports as many possible word-writes as the self-sufficient stage.

The Case of General n

The coding ideas and constructions discussed in Algorithm 1 can be generalized to an arbitrary number of n. Suppose we have $n > 3$ q-level cells to represent $k = 2$ binary variable bits. The idea is to divide the n physical cells in 3 groups, each representing one "virtual cell," and then apply the previous $(3,2)_q$ code. For example, if we have $n = 6$ 4-level cells, we can combine every 2 cells, and make the flash memory act like three virtual cells of 6-level each.

If one has *a priori* knowledge about what word-writes are more possible, then the three groups may be arranged unequally to reflect the application needs. For example, if the application tends to change the first variable more often than the second, then a larger group may be formed for the first virtual cell. In general, such knowledge is either unavailable, or all the three kinds of word-writes tend to be equally probable. Further, considering the fact that

when a group (super cell) is exhausted, the other two can always come to rescue, it is therefore reasonable to evenly divide the cells into groups. When n is not divisible by 3, the surplus cell(s) may either join some of the groups, or be used altogether by the two variable bits to indicate value change; see Algorithm 2.

Theorem 4.2.8. The $(n, 2)_q$ code described in Algorithm 2 guarantees $t = 2m(q-1) + \lfloor \frac{pq}{4} \rfloor$ word-writes of any type, where $n = 3m + p$, and $p = 0, 1, 2$.

Proof. From Theorem 4.2.7, the $(3, 2)_{m(q-1)}$ code guarantees $2m(q-1)$ word-writes. The additional p surplus cells support $\lfloor q/4 \rfloor$ word-writes for $p = 1$ and $\lfloor q/2 \rfloor$ word-writes for $p = 2$. □

Comparison with the existing flash codes: The bit-write optimal flash codes proposed in literature do not come close to our design in terms of guaranteed word-writes. For example, the floating codes in [46] and the multidimensional flash codes in [54] (at $k = 2$) both guarantee about $\frac{1}{2}n(q-1)$ word-writes. In comparison, our $(n, 2)_q$ WEBE codes promise about $\frac{2}{3}n(q-1)$ word-writes, a 33% increase in worst-case performance. On the other hand, our codes fall short in terms of guaranteed bit-writes (except the case $n = 3, q = 2$). Our codes guarantee $\frac{2}{3}n(q-1)$ bit-writes, whereas the bit-write optimal codes guarantee close to $n(q-1)$ bit-writes.

Algorithm 2: $(n, 2)_q$ WEBE codes

1. Suppose $n = 3m + p$, $p = 0, 1, 2$. Evenly divide the last $3m$ cells into three groups, each of which contains m q-level physical cells and can be used to mimic a $m(q-1)$-level virtual cell.

2. Apply the $(3, 2)_{m(q-1)}$ WEBE code discussed in Algorithm 1 on these $3m$ cells.

3. When the these $3m$ cells can no longer support a requested word-write, saturate all of them and go to the remainder p surplus cells. If $p = 1$, then this one extra cell with levels $0, 1, 2, 3, 4, \cdots, q-1$ can be used to represent variable values $(0, 0)$, $(0, 1)$, $(1, 0)$ $(1, 1)$, $(0, 0)$, and so on, and can therefore support $\lfloor q/4 \rfloor$ additional word-writes of any type. If $p = 2$, then these two physical cells are used to represent the two variable bits in the natural way. That is, (x_1, x_2) represents (u_1, u_2), where $u_1 = mod(x_1, 2)$, and $u_2 = mod(x_2, 2)$. These two extra cells can support $\lfloor q/2 \rfloor$ word-writes of any type.

4.2.3 Design WEBE Codes for General k

In previous section, the WEBE codes represented two binary bits are shown and discussed. The upper-bounds of bit-writes and word-writes are asymptotically optimal. Now, we provide the extension idea for WEBE codes to construct general binary bits k for any value of n and q. In this work, we introduce two types of cells as following definitions.

Definition 4.2.9. The *data-state cell* is a cell of flash memory stored the

data that will be represented by the variable bit in the corresponding position. The number of groups(units) of data-state cells is equal to the number of groups(units) of variable bits.

Definition 4.2.10. The *parity-state cell* is a cell of flash memory stored the information from XORing the corresponding multiple variable bits that are updated at the same time to increase the state of all corresponding cells.

Since in this case, we assign both kinds of cells to store data, but the difference from the previous work is that instead of using a virtual cell to rescue and support a large group as soon as it is saturated, we apply the parity-state cells to be like the redundancy in the traditional error-correcting codes. The parity-state cells are computed by XORing any a variable bits, where $0 \leq a \leq k$. The general case of parameters n and k will be discussed in section 4.2.3.

The Simply Case of $(6,3)_2$ WEBE code

Before we extend and construct the WEBE code for general k, in this section we will start showing the simply case of $(6,3)_2$ WEBE code. The layout structure of this code illustrate in Fig.4.6, where we have 3 data-state cells and 3 parity-state cells. The parity-state cells will increase every time there are 2 bits changed/updated at a time. For example, the x_4 stores the XORed data of bits u_1 and u_2, so when u_1 and u_2 are flipped/updated at the same time, instead of increase both x_1 and x_2, we only increase x_4 by 1.

All stored information will be represented by 3 variable bits and Algorithm 3 shows and concludes this code construction both in decoding map and transition map.

Algorithm 3: A class of $(6,3)_q$ WEBE codes:

Notation:

$\mathbf{x}=(x_1,x_2,x_3,x_4,x_5,x_6)$: the state vector, where $x_i=0,1,...,q-1$.

$\mathbf{u}=(u_1,u_2,u_3)$: the variable vector, where $u_i=0$ or 1.

Starting state: $\mathbf{x}=(0,0,0,0,0,0)$, $\mathbf{u}=(0,0,0)$.

Data-state cells: $x_1=u_1, x_2=u_2, x_3=u_3$

Parity-state cells: $x_4=u_1\oplus u_2, x_5=u_2\oplus u_3, x_6=u_1\oplus u_3$

Weight of branch: $W_i=\sum_{i=1}^{n} x_i$

$$W_{u_1} = min(x_1, x_2+x_4, x_3+x_6, x_2+x_5+x_6, x_3+x_4+x_5), \tag{4.6}$$

$$W_{u_2} = min(x_2, x_1+x_4, x_3+x_5, x_1+x_5+x_6, x_3+x_4+x_6), \tag{4.7}$$

$$W_{u_3} = min(x_3, x_1+x_6, x_2+x_3, x_1+x_4+x_5, x_2+x_5+x_6) \tag{4.8}$$

Note that x_i must not be a full cell to be considered as a subset of minimum weight.

Decoding Map:

$$u_1 = x_1\oplus x_4\oplus x_6, \tag{4.9}$$

$$u_2 = x_2\oplus x_4\oplus x_5, \tag{4.10}$$

$$u_3 = x_3\oplus x_5\oplus x_6 \tag{4.11}$$

where $\oplus$ denotes binary addition or XOR.

Transition Map:

- $(u_1,u_2,u_3)\rightarrow(u_1\oplus 1,u_2,u_3)$:
  Increase x_1 by 1 if possible; otherwise, increase the 2 or 3 related minimum-weight cells by 1.
- $(u_1,u_2,u_3)\rightarrow(u_1,u_2\oplus 1,u_3)$:
  Increase x_2 by 1 if possible; otherwise, increase the 2 or 3 related minimum-weight cells by 1.
- $(u_1,u_2,u_3)\rightarrow(u_1,u_2,u_3\oplus 1)$:
  Increase x_3 by 1 if possible; otherwise, increase the 2 or 3 related minimum-weight cells by 1.
- $(u_1,u_2,u_3)\rightarrow(u_1\oplus 1,u_2\oplus 1,u_3)$:
  Increase x_4 by 1 if possible; otherwise, increase the 2 or 3 related minimum-weight cells by 1.
- $(u_1,u_2,u_3)\rightarrow(u_1,u_2\oplus 1,u_3\oplus 1)$:
  Increase x_5 by 1 if possible; otherwise, increase the 2 or 3 related minimum-weight cells by 1.
- $(u_1,u_2,u_3)\rightarrow(u_1\oplus 1,u_2,u_3\oplus 1)$:
  Increase x_6 by 1 if possible; otherwise, increase the 2 or 3 related minimum-weight cells by 1.
- $(u_1,u_2,u_3)\rightarrow(u_1\oplus 1,u_2\oplus 1,u_3\oplus 1)$:
  Increase 2 related minimum-weight cells by 1 if possible; otherwise, increase the 3 related minimum-weight cells by 1.

Example 4: Consider using $(6,3)_2$ WEBE code in 6 $q-$level cells to represent a variable word of 3 bits where there are 3 data-state cells, 3 parity-state cells and their relations are shown in Fig.4.6. Let $q = 2$, and Fig. 4.7 presents a graph illustration when cells are written from an empty state. From Algorithm 3, using the minimum-weight selection method to choose which cells we need to update after the bit is updated. The number of word-writes will be increased and asymptotically optimal.

The Case for General k

To extend the technique of constructing WEBE code in the previous work for general k, we have introduced the *parity-state cell* which can be computed and defined by XOR operations. In this work, we let the number of total cells be n which are separated into k data-state cells and $n - k$ parity-state cells, where $k \leq n$. These cells store information and are represented k variable bits.

However, to construct the WEBE code for arbitrary k and n, there are various possibilities to generate the parity-state cells. The maximum and optimal number of parity-state cells are $m = n - k = 2^k - 1$. Also, the number of data-state cells that the parity-state cells have to be covered is not fixed. We can assign 2,3,4, or more data-state cells to be XORed and represented in one parity-state cells that can return and transform the updated data keeping in such cell into the updated variable bits (or let's say a bits to be XORed, where $0 \leq a \leq k$). This technique will help to extend the time to erase the whole block of cells when one cell is full and cannot be increased to the higher state any more, since there are always the parity-state cells that cover/backup such cell and we can write/update the information into them.

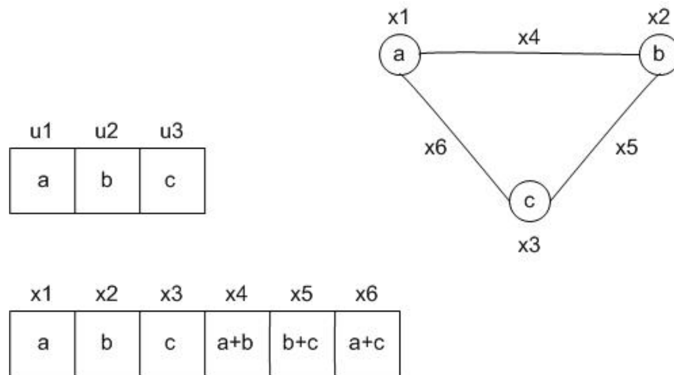


Figure 4.6: An example of a simple $(6,3)_q$ WEBE code

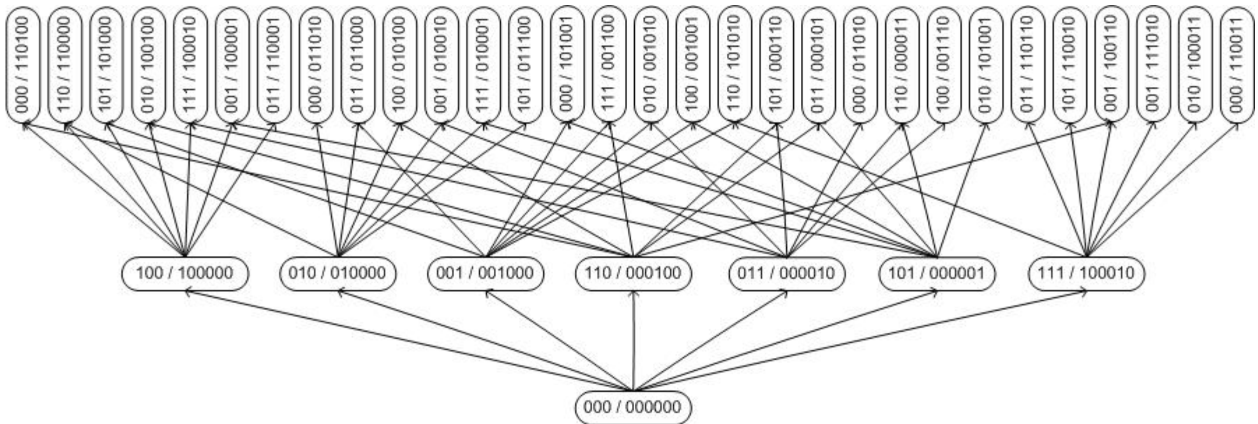


Figure 4.7: A $(6,3)_2$ WEBE code that achieve an asymptotically optimal word-writes

Example 5: Shown in Fig. 4.8 is one example of various layout structures to construct an $(n, k)_q$ WEBE code. This layout illustrates that there are k variable bits to represent n $q-$level cells, where there are k data-state cells and $n - k$ parity-state cells. From a graphical structure, each parity-state cell is computed by XORing any 2 adjacent bits ($a = 2$), since we use a cycle graph to define which cells are written when some sets of variable bits is updated. The edges of a graph represent each parity-state cell that can be computed by XORing two data-state cells at both ends.

Algorithm 4 shows the concise and rigorous mathematical definition and process to construct an $(n, k)_q$ WEBE code.

Algorithm 4: A class of general $(n,k)_q$ WEBE codes:

Notation:

$\mathbf{x}=(x_1, x_2, ..., x_n)$: the state vector, where $x_i=0, 1, ..., q-1$.

$\mathbf{u}=(u_1, u_2, ..., u_k)$: the variable vector, where $u_i=0$ or 1, and $k \leq n$, $n=k+m$.

Starting state: $\mathbf{x}=(0, 0, ..., 0)$, $\mathbf{u}=(0, 0, ..., 0)$.

Data-state cells (k): $x_1=u_1, x_2=u_2, ..., x_k=u_k$

Parity-state cells (m): $x_{k+1}=u_1 \oplus u_2, x_{k+2}=u_2 \oplus u_3, ..., x_n=u_k \oplus u_1$

(Note that this scheme is only one subclass of all possibilities to construct $(n,k)_q$ WEBE codes. We can XOR a sets of bits, where $0 \leq a \leq k$).

Weight of branch: $W_i = \sum_{i=1}^{n} x_i$

(Note that x_i must not be a full cell to be considered as a subset of minimum weight.)

Decoding Map:

$$u_i = x_i \oplus x_j \oplus x_l, \tag{4.12}$$

where x_j and x_l denote the cells that related to cell x_i (two connecting edges of x_i.)

Transition Map:

- 1 bit changed u_i, $i \in 1, 2, ..., k$:
  Increase x_i by 1 if possible; otherwise, increase the 2 or more related minimum-weight cells by 1.

- 2 bits changed (u_i, u_j):
  Increase the cell stored $u_i \oplus u_j$ by 1 if possible; otherwise, increase the 2 or more related minimum-weight cells by 1.

- more than 2 bits changed (m bit changed):
  Increase the least number of cells that related to all changed bits and has minimum weight by 1 if possible; otherwise, increase more cells that can be XORed and covered all changed bits by 1.

The $(n, k)_q$ WEBE code will be optimal if and only if the number of cells $n = 2^k - 1$. Nevertheless, the simulation results in Fig. 4.9 show the number of word-writes for $(6, 3)_q$, and $(5, 3)_q$ WEBE codes. Additionally, the optimal one (from $(7, 3)_q$ WEBE code) also shown in this graph.

Fig. 4.9 also shows the comparison between the random selection method when we randomly select any subset of cells to rewrite/update into flash memory, and the minimum-weight selection method when we use for this WEBE code. Clearly, the minimum-weight selection method outperforms the random one and the curve is closed to the optimal $(7, 3)_q$ WEBE code when we remove 1 cell from the optimal case. However, in general case the $(n, k)_q$ WEBE codes are flexible to construct so that the number of parity-state cells (m) is not fixed and the maximum of m is $m = (2^k - 1) - k$.

It should be noted that the case of general $(n, k)_q$ WEBE code shown in Fig.4.8 is only a subsume of all possibilities to construct this code. In practical, we can flexible construct the parity-state cells by XORing $2, 3$, or any k data-state cells. In future work, we can extend this code to detect and correct some errors that may occur during data processing.

4.3 Flash Marker (FM) Codes

In this section, we propose a new code termed the "*Flash Marker (FM)*" code for arbitrary n, k, and q applied for the strategy that there is some cell stored often-updated data to be more suitable for the practical use, thus this cell has the highest probability among all cells to be written and be the first one to reach the highest cell state level ($q_i = q - 1$). Our code will provide and reserve spare cells for this frequently used cell to lengthen the time to reset

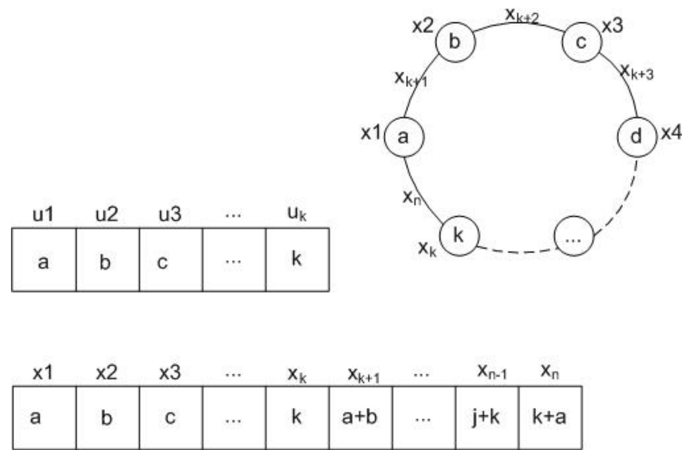


Figure 4.8: One example of layout structures of $(n, k)_q$ WEBE code

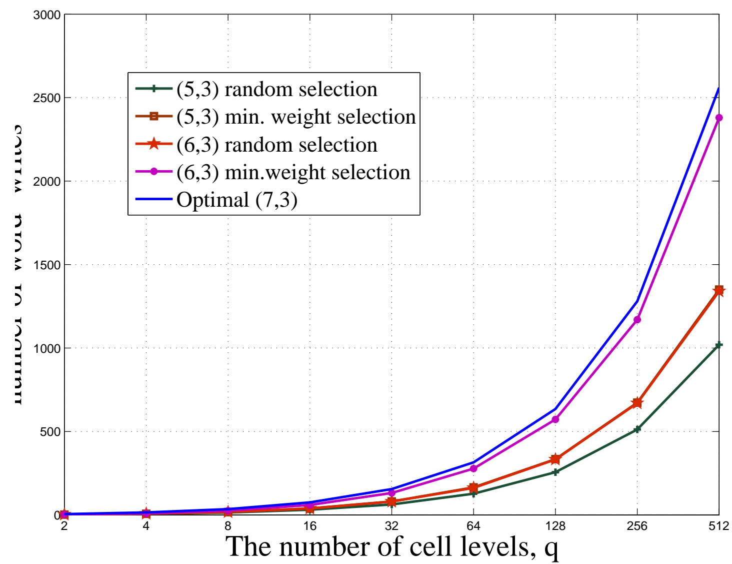


Figure 4.9: The number of word-writes $(5, 3)_q$ and $(6, 3)_q$ WEBE codes for the various value of q.

the whole block of cells. Our goal is to maximize the writing times– the writes we consider here are both bit-writes and word-writes defined in the previous work when the value of at least one of K data bits is changed.

The motivation of this code is in practical, the uneven writings, which some data may be updated more frequently than others, are usually happened. Then, the worst case will occur when the most updated bit (and its corresponding cells) has exhausted its states, even there may be many other unexhausted cells in the block, so that the whole block will be reset. Also, there is no means of knowing what bits will be updated beforehand, so a fixed, uniform resource allocation is not optimal. Thus, a run-time on-demand resource allocation is desired to extend the time to reset the cell block and try to efficiently use all cells before block erasure.

4.3.1 FM Code Construction

To construct $(N, K, s)_q$ FM code, we design the system model as the assumptions shown following. Note that the total number of cells is $N = (k + s)n$, where k is the number of fixed-assignment of cell/variable bit-pair units with n cells each, and s is the number of spare-cell units (the on-demand assignment). Thus, the number of total variable bits is $K = 2k$.

Assumptions and Notations:

1. k units(groups) of data cells and s units of spare cells in any flash memory
2. In each unit, there are n cells and each cell has q states (for q is an even integer)
3. In each unit, each cell can be written either from left side to right side or vice versa, when there are only any 2 consecutive cells left in each

block that still have some level to increase, we have to consider which cell is the next to write and which cell will be an marker cell which we will explain later.

4. Each bit pair in variable vector is represent the value of cell in each correspondent unit in cell, so the totally number of variable bits $K = 2k$.

5. If the marker cell reach the lowest state of marker state(state ii^{th}) which is $q_{ii} = q - s$, it points to the spare cell m_{ii} and start writing in that cell.

Consideration of marker states of marker cell

- If there are only 2 consecutively active cells available to update and either cell is empty, this unit can be updated from both left or right sides.
- If there are only 2 consecutively active cells available to update, one cell has lower state level than the other, and the higher state cell is at the state $q - s$, this unit can write new data in the lower one and the higher cell can be an marker cell and start counting as an marker state if it can point to spare cells which are still available to write.
- If there are only 2 consecutively active cells available to update, one cell has lower state level than the other, the higher state cell is higher than the state $q - s$, and the lower on is lower than or at state $q - s$, this unit can write new data in the lower one and the higher cell is an marker cell and already counting as a marker state if it can point to spare cells which are still available to write. However, if spare cells are unavailable and already reserved by another marker cell, this cell can be written until it reaches the highest state, $q - 1$.
- If there is only one active cell available to update, this unit can update its data by increasing the state level of this cell and this cell is called a

marker cell, then it will start writing on spare cells as soon as it reaches the state $q-s$ and spare cells are available to write.

In each unit we can represent its stored data as 2 variable bits. Let $x = \{x_0, x_1, \cdots, x_{k-1}\}$ be the information storing in each cell in any unit, $m = \{m_0, m_1, \cdots, m_{s-1}\}$ be the spare-cells unit that are the extension of s marker states of the marker cell from any group that can first access to this spare cell (or the first unit that fills up the marker states) as shown in Fig.4.10. Note that the number of marker states is equal of the number of spare cells.

Encoding or Transition map ($x_K \to v_K$):

1. In each unit of cells, we start writing data into flash memory from either left- or right-edge of cell. As soon as any unit writes on the marker cell and reaches the spare cell unit, other units cannot use and access to that spare cell unit.
2. At the s highest states of marker cell of the first written unit, $x_m = \{x_{q-s}, x_{q-s+1}, \cdots, x_{q-2}, x_{q-1}\}$ will be indicated to the s spare cells where $x_{q-s} \to m_0, x_{q-s+1} \to m_1, \cdots, x_{q-1} \to m_s$.

Decoding map ($v_K \to x_K$):

1. From any $v_K = \{v_1, v_2, v_3, \cdots, v_K\}$, where K is an even integer, $K = 2k$, we can group this variable vectors as k groups, so we also have k sets of a bit pair.
2. In each group, the decoding process is the same as the previous section.

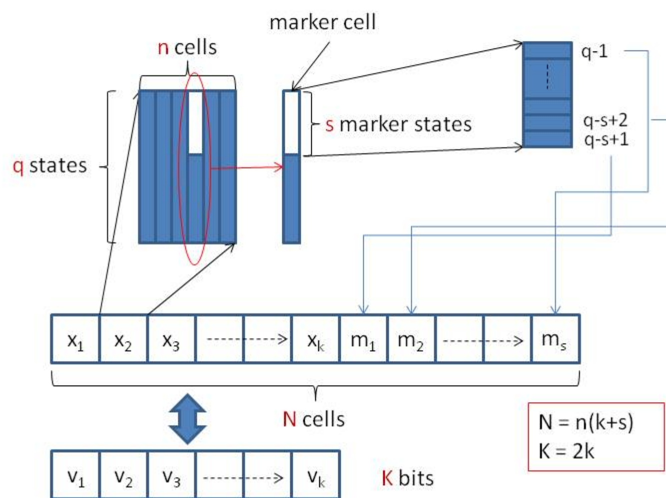


Figure 4.10: The relation of s marker states, s spare cells of $(N, K, s)_q$ FM code

Therefore, both transition map and decoding map are given in the concise and rigorous mathematical definition and process as shown in Algorithm 5 to construct an $(N, K, s)_q$ FM code.

Algorithm 5: A class of $(N, K, s)_q$ FM codes:

Notation:

$\mathbf{x} = (x_0, x_1, ..., x_{n-1}|, x_n, x_{n+1}, ..., x_{2n-1})|, ...|, x_{(k-1)n}, x_{(k-1)n+1}, ..., x_{kn-1})$: the cell-state vector, where $x_i = 0, 1, ..., q-1$.

At any unit jth, all n cells can be separated into 3 groups: x_1, x_2, x_3, where x_3 is a marker cell.

$\mathbf{m} = (m_0, m_1, ..., m_{n-1}|, m_n, m_{n+1}, ..., m_{2n-1})|, ...|, m_{(s-1)n}, m_{(s-1)n+1}, ..., m_{sn-1})$: the spare-state vector, where $m_i = 0, 1, ..., q-1$.

$\mathbf{u} = (u_1, u_2|, u_3, u_4|, ...|, u_{2k-1}, u_2k)$: the variable vector, where $u_i = 0$ or 1.

Starting state: $\mathbf{x} = (0, 0, 0, ..., 0)$, $\mathbf{m} = (0, 0, 0, ..., 0)$, $\mathbf{u} = (0, 0, 0, ..., 0)$.

Decoding Map: At any unit jth of variable vector and cell-state vector, where $j = 1, 2, ..., k$ and $x_j = x_1, x_2, x_3$:

$$u_{2j-1} = mod(x_1, 2) \oplus mod(x_3, 2), \quad (4.13)$$

$$u_{2j} = mod(x_2, 2) \oplus mod(x_3, 2), \quad (4.14)$$

where $\oplus$ denotes binary addition (i.e. exclusive OR, XOR), and $mod(x_j, 2)$ is a modulo 2 operation.

Transition Map:

- $(u_{2k-1}, u_{2k}) \to (u_{2k-1}+1, u_{2k})$:
  Increase x_1 by 1 if possible, then, increase both x_2 and x_3 by 1; otherwise increase m_{ii}.

- $(u_{2k-1}, u_{2k}) \to (u_{2k-1}, u_{2k}+1)$:
  Increase x_2 by 1 if possible, then increase both x_1 and x_3 by 1; otherwise, increase m_{ii}.

- $(u_{2k-1}, u_{2k}) \to (u_{2k-1}+1, u_{2k}+1)$:
  Increase x_3 by 1 if possible, then, increase both x_1 and x_2 by 1; otherwise increase m_{ii}.

Note that m_{ii} is the iith spare-cell unit when x_3 reaches the area of marker states ($x_3 \geq q - s$).

Example 6: Let $n = 5, q = 4$, then $q_{ii} \in \{0, 1, 2, 3\}$ in each cell. Let $k = 2, s = 1$, then the number of cells is $N = n(k+s) = 15$ cells The parameters of variable bits are $K = 2k, l = 2$, so that $l_j \in \{0, 1\}$. If we begin with the empty state where all cells have not been written yet and contain all 0's, one of simple ways to represent the data that will be written in cells is shown below.

Cell state: 00000 00000 00000 → 10000 00001 00000 → 20000 00001 00000 → 30000 00001 00000 → 31001 00001 00000 → 32002 00002 00000 → 33003 10002 00000 → 33103 20002 00000 → 33213 30002 00000 → 33323 30002 00000 → 33333 31002 00000 → 33333 31003 10000 → 33333 32013 20000.

Variable bit: 00 00 → 10 01 → 00 01 → 10 01 → 01 01 → 10 00 → 01 10 → 11 00 → 00 10 → 11 10 → 10 00 → 11 01 → 10 10.

Fig. 4.11 shows some stages of the cell-state updates in *Example 6*, and it is clearly shown that a spare cells will be written when the first group of cells is full (since this group is more frequently updated than the others). So, instead of erasing the entire block as soon as the first group is full/saturated, this block of flash memory still has some available cell (both spare cells and cells in the other group) to be written.

4.3.2 Simulation Results

To consider the number of bit-writes compared to the number of word-writes, we have simulated $(N, K, s)_q$ FM codes under the assumptions which are mentioned in the previous section. The results are represented in the bar graphs in Fig. 4.12 and Fig. 4.13.

Fig. 4.12 shows the number of bit-writes of $(N, K, s)_q$ FM codes. The

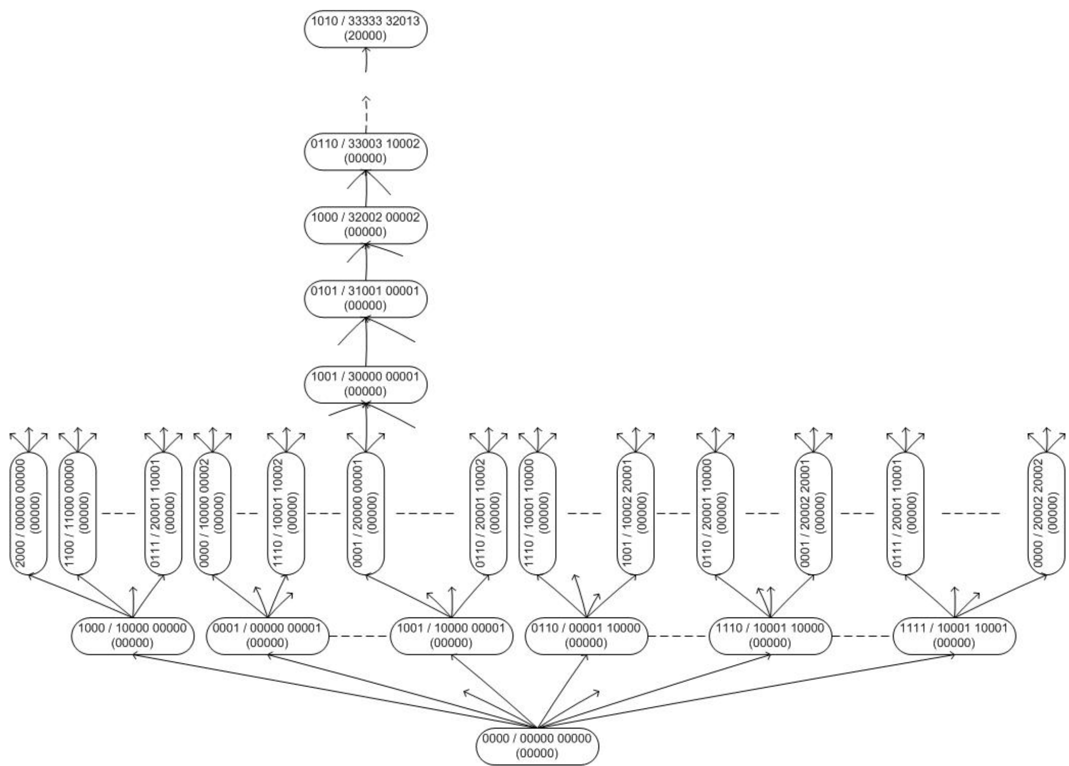
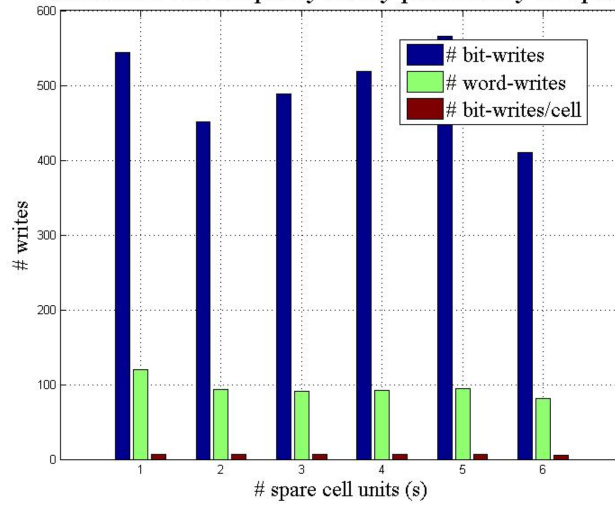
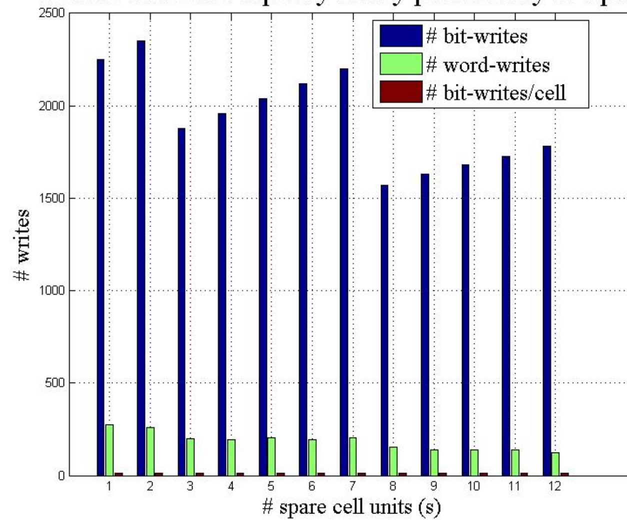


Figure 4.11: An example of cell-state updates of $(15, 4, 1)_4$ FM code shown in *Example 6* (all cells shown in the parentheses are spare cells).



(a) The number of bit-writes when $k = 8, q = 8$, and all bits have equally likely probability to update



(b) The number of bit-writes when $k = 16, q = 16$, and all bits have equally likely probability to update

Figure 4.12: The number of bit-writes of $(N, K, s)_q$ FM codes when the number of spare-cell units (s) is increased

number of bit-writes of FM codes with an equally likely probability to update bits when the number of spare cell units is increased for both ($q = 8$) and ($q = 16$) flash memories are shown in Fig.4.12(a) and Fig.4.12(b), respectively. The results show that the number of bit-writes is still depended on the number of cells and the number of bit-writes per cell is almost constant while the number of spare cells (s) is increasing.

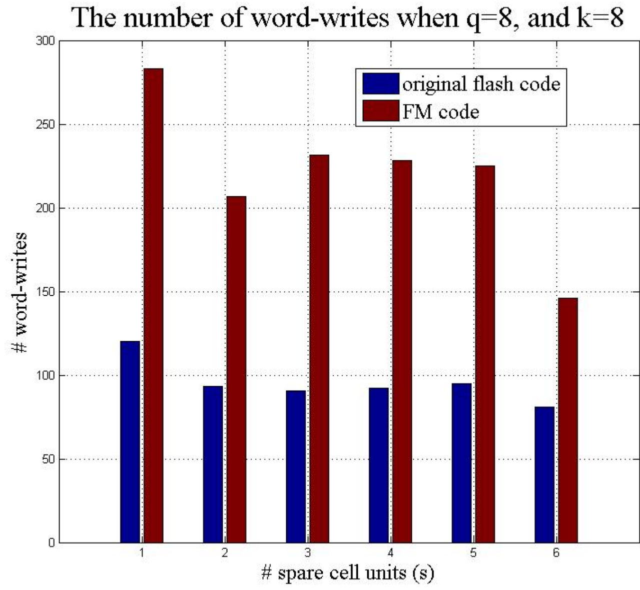
In Fig.4.13 the interesting result is that the more the number of spare cell units, the less the number of word-writes. So, we consider the case of an unequally weight probability to update bits which we always reserve the spare cell units for cell units that are frequently used/written. The results are shown in Fig.4.13(a) and Fig.4.13(b) for $q = 8$ and $q = 16$, respectively. Clearly, we have more word-writes when we know which cell units need spares and always reserve one for them.

4.3.3 Discussion

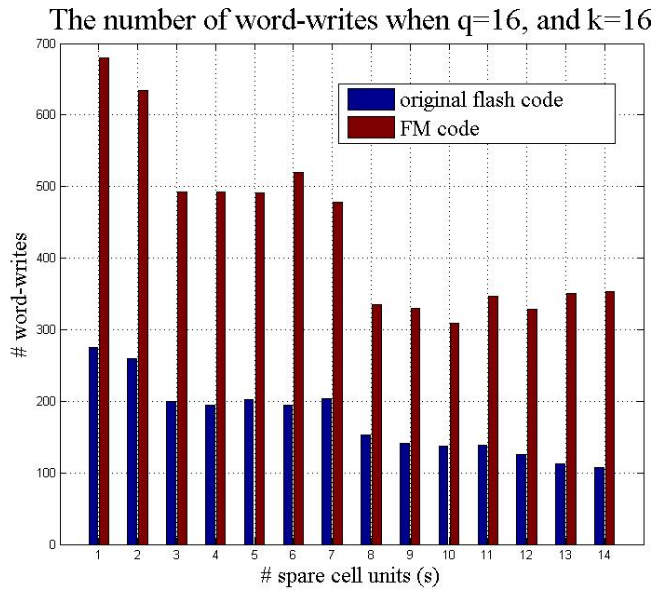
To consider the write deficiency, the optimal number of writes from our codes is $N(q-1)$ when N, is large. We can see that our codes are also asymptotically optimal codes.

Theorem 4.3.1. If there are N $q-$level cells, where $N = (k + s)n$ with k cell units and s spare cell units, then the FM code can guarantee at least $t = \{(n - 1)(q - 1) + (q - s)\}(k + s)$.

Proof. Consider each unit of cell, at the worst case where both bits are updated at the same time, the left and right side cells are written and increased their levels up to the higher states before they both reach the marker cell simultaneously, so it is at most $t = (n - 1)(q - 1)$ writes. Then, at the marker cell, this cell can be increased up to $q - s$ levels. Thus, in each unit we have



(a) The number of word-writes when $q = 8, k = 8$ with unequally weight flipped bits



(b) The number of word-writes when $q = 16, k = 16$ with unequally weight flipped bits

Figure 4.13: The number of word-writes of $(N, K, s)_q$ FM codes when the number of spare-cell units (s) is increased.

the number of bit writes, $t = (n-1)(q-1)+(q-s)$ and since there are totally $k+s$ units, so this code guarantees $t = \{(n-1)(q-1)+(q-s)\}(k+s)$. $\square$

In conclusion, the FM code, which are a combination of fixed cell allocation and adaptive cell allocation, can achieve asymptotically optimal in terms of the number of writes when applying for the model that we know which cells and which units are frequently updated and always reserve spare-cell units for those cells (non-uniform manner). The update strategy of these codes is more efficient and extends a life cycle. Additionally, we can update two data bits at a time from only changing one bit in cell state vector.

4.4 Conclusion

We have proposed two novel ideas of coding for flash memory. The contributions from this work are: (1) we have introduced and defined the "word-write" to measure how much this code can be support the number of writes from the user's view before the block erasure is needed, (2) the $(n,k)_q$ WEBE codes are simple to construct and proved that they are efficient in terms of bit-writes and achieve more word-writes compared to the existing codes in [46] and [54], and (3) the $(N,K,s)_q$ FM codes are designed for specific applications when there is some file/data bit frequently been updated/written (non-uniform manner). Additionally, other improvements on both codes are the promising area for future works. We can also extend this code for a capability to detect and correct some errors in order to improve the reliability of flash memory.

Chapter 5

Summary and Future Works

This dissertation investigates an erasure-correction code technology in the area of data storage including in disk arrays, large-scale data storage systems, and flash memories.

With the explosive increase of digital data everyday, data storage is becoming the center of today's cyber infrastructure. Due to the high competition in industry, the desirable data storage solution should have high capacity, reliability, speed in write/read operations, and low overhead. To improve the performance of data storages, we have proposed the coding techniques applied on both disk drives and flash drives in different perspectives. In disk drives, our goal is to protect and recover all disks from data loss due to disk failures. The designed codes should have ability to recover and handle disk failures in an effective and efficient way. In flash drives, we focus on lengthening their lifespan by using coding techniques to maximize the number of writes before a block of cells need to be erased. The work presented in this dissertation also set the corner stone for future generalization and extension.

5.1 Data Disks

For the coding techniques applied on data disks, we have investigated new ways of generating MDS array codes. We have directly applied the graphical representation called the *complete-graph-of-ring (CGR) graph* to construct the CGR array codes which are optimal MDS codes. Their dual codes are also MDS. These codes have low decoding/updating complexity and simple implementation that uses only the XOR operations.

Additionally, we can consider this CGR array code as a modified LDPC code. We have defined the difference between our code with the traditional LDPC codes. Erasures can be recovered by using row- and column-operations with efficiency. The code is suitable for direct-decoding in distributed storage systems.

In the future, one can research a longer MDS array code for a larger data network with different graph models. It is beneficial to study a generalized form of compound graph codes with more flexibility and more choices for rates and sizes. Constructing MDS array codes in terms of the parity-check matrix H and generator matrix G, especially in the form of quasi-cyclic LDPC matrix, is also a promising research direction.

5.2 The Distributed Storage Networks

For a large-scale data center/network, we have investigated coding techniques to help recover and protect data loss that may occur from several reasons. We especially focused on data loss caused by broken/failed data disks. Here, we concatenated optimal MDS codes and LT codes, called "*the MDS-LT nested*

codes," to provide a larger erasure correcting capability. Moreover, to ease the implementation and reduce encoding/decoding complexity we constructed them in hierarchical protection with local, regional, and global parity disks. The overall code is not an MDS code, and still leaves room for development and improvement.

In addition, we have also proposed the fixed, rigid structure to construct layered erasure codes by applying a set of MDS codes for local protection and then protected by the higher protection of two-dimensional SPC code. This code, namely " *the horizontal-vertical single parity check (HVSPC) code*," is easy to implement and flexible to construct for different number of erasures. Comparison with the MDS-LT nested codes, the HVSPC codes require less overhead. The overall code is nevertheless not MDS.

For future work, we intend to consider other structures of nested codes and MDS codes to achieve the space optimality as much as possible. We will also consider codes with more than two layers, as well as unequal error protection (UEP).

5.3 Flash Memories

Flash memory is an emerging data storage technology that may eventually replace all disk drives in the near future. Recently, most of the research of coding techniques for flash memory are published in U.S. patent documents. This research field is expected to receive more attention in the future.

The downside of flash memory is the limited number of writes before a block erasure (aka block reset) must occur, resulting in a shorter lifespan of this flash memory. We developed the WEBE codes, first for $n = 3$ $q-$states

cells and $k = 2$ data bits, and then for the general case with an arbitrary n and k. The algorithms and methods developed here will likely find useful applications in an area not only for data storage, but also for data accessing organization.

In addition, we have developed the flash marker (FM) codes specially designed to address the issue when data do not have the same probability to be written. FM codes will adaptively assign spare cells for the cells storing frequently-updated data. The marker states of a marker cell connects to spare cells. This code can increase the number of word-writes as shown in the simulation results.

In both codes we have developed, we have first introduced the number of *word writes* to measure the performance of our code instead of counting only the number of bit writes.

For future work, one can apply error-correcting codes to both represent stored data in flash memory and correct some errors that may happen during data processing. It is also fruitful to extend and improve the WEBE codes to be able to correct some errors since it has 2 types of cells: data-state cells and parity-state cells, like a traditional error-correcting code that we use to detect and correct errors.

Bibliography

[1] B. Vasic, and E. M. Kurtas, *Coding and Signal Processing for Magnetic Recording Systems,* CRC Press, 2004.

[2] R. Micheloni, A. Marelli, and R. Ravasio, *Error Correction Codes for Non-Volatile Memories*, 2008.

[3] L. Xu, V. Bohossian, J. Bruck, and D. G. Wagner, "Low-Density MDS Codes and Factors of Complete Graphs," *IEEE Trans. on Information Theory*, vol.45, pp.1817-1826, Sept. 1999.

[4] J. S. Plank, "A Tutorial on Reed-Solomon Coding for Fault-Tolerance in RAID-like Systems," Software Practice and Experience, 27(9), pp.995-1012, Sept. 1997.

[5] A. Dholakia, E. Eleftheriou, X. Yu, I. Iliadis, J. Menon, and K. Rao, "A New Intra-Disk Redundancy Scheme for High-Reliability RAID Storage Systems in the Presence of Unrecoverable Errors," *ACM Trans. on Storage,* pp.1-42, May 2008.

[6] M. Blaum, P. Farrell, and H. van Tilborg. Array codes. "Handbook of Coding Theory," V.S. Pless and W.C. Huffman, pp.1805-1909.

[7] M. Blaum, J. Brady, J. Bruck, and J. Menon, "EVENODD: An Efficient Scheme for Tolerating Double Disk Failures in RAID Architectures," *IEEE Trans. on Computers*, vol.44, pp.192-202, Feb. 1995.

[8] L. Xu, and J. Bruck, "X-Code: MDS Array Codes with Optimal Encoding," *IEEE Trans. on Information Theory*, vol.45, pp.272-275, Jan.1999

[9] J. L. Hafner, "HoVer Erasure Codes for Disk Arrays," *IBM Research Report*, Almaden Research Center, July 2005.

[10] J. L. Hafner, V. Deenadhayalan, and KK Rao, "Matrix Methods for Lost Data Reconstruction in Erasure Codes," *FAST'05:4th USENIX Conference on File and Storage Technologies,* pp.183-196, 2005.

[11] J. L. Hafner, V. Deenadhayalan, T. Kanungo, and KK Rao, "Performance Metrics for Erasure Codes in Storage Systems," *IBM Resaerch Report,* Almaden Research Center, Aug. 2004.

[12] W.D. Wallis, *One-Factorization*, Norwell, MA: Kluwer, 1997.

[13] Y. Cassuto, "Coding Techniques for Data-Storage Systems," Thesis, California Inst. of Tech., Dec. 2007.

[14] Y. Cassuto, and J. Bruck, "Array Codes for Clustered Column Erasures," *ISIT*, pp.1726-1730, July 2008.

[15] Y. Cassuto, and J. Bruck, "Cyclic Lowest Density MDS Array Codes," *IEEE Trans. on Information Theory,* pp.1721-1729, Apr. 2009.

[16] C. M. Kozierok, "Redundant Arrays of Inexpensive Disks," *The PC Guide*, Pair Networks, April, 2001.

[17] D. A. Patterson, G. Gibson, and R. H. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID,)" *Proceeding ACM SIGMOD,* pp.109-116, June 1988.

[18] P. Guide, "Multiple (Nested) RAID Levels," *http://www.pcguide.com/ref/hdd/perf/raid/levels/mult.html*, Apr. 2007.

[19] M. S. Manasse, C.A. Thekkath, and A. Silverberg,“A Reed-Solomon Code for Disk Storage, and Efficient Recovery Computations for Erasure-Coded Disk Storage,” *Proceeding in Informatics,* pp.1-11, Available at: http://research.microsoft.com/pubs/64690/wdas.pdf

[20] R. Gallager, “Low-Density Parity Check Codes,” *IRE Trans. on Information Theory,* pp.21-28, Jan. 1962.

[21] H. Kaneko, and E. Fujiwara, “Reconstruction of Erasure Correcting Codes for Dependable Distributed Storage System without Spare Disks,” *IEEE 22nd International Symposium on Defect and Fault Tolerance in VLSI Systems,* pp.349-357, 2007.

[22] A. Shokrollahi, “Raptor Codes,” *IEEE Trans. on Information Theory,* pp.2551-2567, June 2006.

[23] P. Kaewprapha, N. Puttarak, and J. Li, “Nested Erasure Codes to Achieve the Singleton Bounds,” *Proc. CISS,* 2009.

[24] N. Puttarak, P. Kaewprapha, and J. Li, “A New Class of MDS Erasure Codes based on Graphs,” *IEEE GlobeCom,* 2009.

[25] M. Luby, “LT Codes,” *Proceeding of the 43rd Annual IEEE Symposium Foundations of Computer Science,* 2002.

[26] P. Cataldi, M. P. Shatarski, M. Grangetto, and E. Magli, “Implementation and Performance Evaluation of LT and Raptor Codes for Multimedia Applications,” *Intelligent Information Hiding and Multimedia Signal Processing,* pp.263-266, Dec. 2006.

[27] M. Luby, M. Mitzenmacher, A. Shokrollahi, and D. Spielman, “Efficient Erasure Correcting Codes,” *IEEE Trans. on Information Theory,* pp.569-584, Feb.2001.

[28] R. Karp, M. Luby, and A. Shokrollahi, “Finite Length Analysis of LT Codes,” *ISIT 2004,* June 2004.

[29] B. Gaidioz, B. Koblitz, and N. Santos, “Exploring High Performance Distributed File Storage Using LDPC Codes,” *Elsevier,* Jan. 2007.

[30] J. S. Plank, and M.G Thomason, “On the Practical Use of LDPC Erasure Codes for Distributed Storage Applications,” Sept.2003.

[31] White paper, “NAND vs. NOR Flash Memory Technology Overview,” Toshiba.

[32] A. Thomasian, and M. Blaum, “Higher Reliability Redundant Disk Arrays: Organization, Operation, and Coding,” *ACM Transaction on Storage*, vol.5, Nov.2009.

[33] L. Hellerstein, G. A. Gibson, R. M. Karp, R. H. Katz, and D. A. Patterson, “Coding Techniques for Handling Failures in Large Disk Arrays,” *3rd International conference of Architectural Support for Programming Languages and Operating Systems (ASPLOS III),* March, 1989.

[34] M. Schulze, G. Gibson, R. Katz, and D. Patterson, “How Reliable is a RAID?,” *IEEE,* 1989.

[35] C. Huang, and L. Xu, “STAR: An Efficient Coding Scheme for Correcting Triple Storage Node Failures,” *IEEE Trans. on Computers*, pp.889-901, July 2008.

[36] J. Lacan, and J. Fimes, ”Systematic MDS Erasure Codes Based on Vandermonde Matrices,” *IEEE Commu. Letters*, vol.8, pp. 570-572, Sept. 2004.

[37] P. Corbett, B. English, A. Goel, T. Grcanac, S. Kleiman, J. Leong, and S. Sankar, “Row-Diagonal Parity for Double Disk Failures,” *FAST’04,* pp.1-14, 2004.

[38] P. Sobe, and K. Peter, “Flexible Parameterization of XOR based Codes for Distributed Storage,” *IEEE Sym. on Network Computing and Applications,* pp.101-110, 2008.

[39] M. Li, J. Shu, and W. Zheng, "Grid Codes: Strip-Based Erasure Codes with High Fault Tolerance for Storage Syatems," *ACM Transactions on Storage,* vol.4, pp.15:1-15:22, Jan. 2009.

[40] M. Li, and J. Shu, "On the Equivalence between the B-Codes Constructions and Perfect 1-Factorization," *ISIT 2010,* pp.993-996, June 2010.

[41] M. Aguilera, R. Janakiraman, and L. Xu, "Using Erasure Codes Efficiently for Storage in a Distributed System," *ICDCS*, pp.106-120, Oct. 2003.

[42] H. Fujita, and K. Sakaniwa, "Modified Low-Density MDS Array Codes for Tolerating Double Disk Failures in Disk Arrays," *IEEE Trans. on Computers,* pp.563-566, Apr. 2007.

[43] P. Cappelletti, C. Golla, P. Olivo, and E. Zanoni *(Ed.), Flash Memories,* Kluwer Academic Publishers, 1st Edition, 1999.

[44] S. K. Lai, "Flash Memories: Successes and Challenges," *IBM J. Res. and Dev.*, Vol.52, pp.529-535, July/Sep. 2008.

[45] A. Jiang, and J. Bruck, "Joint Coding for Flash Memory Storage," *ISIT 2008*, pp.1741-1745. July 2008.

[46] A. Jiang, V. Bohossian, and J. Bruck, "Floating Codes for Joint Information Storage in Write Asymmetric Memories," *ISIT 2007*, June 2007.

[47] R. Rivest, and A. Shamir, "How to reuse a 'write-once' memory," *ACM 1982*, pp. 105-113.

[48] F. Fu, and A. J. Han Vinck, "On the Capacity of Generalized Write-Once Memory with State Transition Described by an Arbitrary Directed Acyclic Graph," *IEEE Trans. on Information Theory,*, pp. 308-313, Jan. 1999.

[49] F. Fu, and R. W. Yeung, "On the Capacity and Error-Correcting Codes of Write-Efficient Memories," *IEEE Trans. on Information Theory,* pp. 2299-2314, Nov. 2000.

[50] R. Ahlswede, and Z. Zhang, "Coding for Write-Efficient Memory," *Information Computer*, 1989.

[51] E. Gal, and S. Toledo, "Algorithms and Data Structures for Flash Memories," *ACM Computing Surveys,* pp.138-163, June 2005.

[52] V. Bohossian, A. Jiang and J. Bruck, "Buffer Coding for Asymmetric Multi-Level Memory", *ISIT 2007*, June 2007.

[53] V. Bohossian, and J. Bruck, "Shortening Array Codes and the Perfect 1-Factorization Conjectures," *IEEE Trans. on Information Theory,* pp.507-513, Feb. 2009.

[54] E. Yaakobi, A. Vardy, P. H. Siegel, and J. K. Wolf, "Multidimensional Flash Codes," *Proc.* 46th *Annual Allerton Conf. on Commu. Control and Computing*, 2008.

[55] E. Yaakobi, P. H. Siegel, A. Vardy, and J. K. Wolf, "Multiple Error-Correcting WOM-Codes," *ISIT 2010,* pp.1933-1937, June 2010.

[56] H. Finucane, Z. Liu, and M. Mitzenmacher, "Designing Floating Codes for Expected Performance," *Proc. 47th Allerton Conf.,* pp.1389-1396, Sept. 2008.

[57] A. Jiang, R. Mateescu, M. Schwartz, and J. Bruck, "Rank Modulation for Flash Memories," *ISIT 2008*, pp.1731-1735, July, 2008.

[58] A. Jiang, M. Schwartz, and J. Bruck, "Error-Correcting Codes for Rank Modulation," *ISIT 2008* , pp. 1736-1740, July 2008.

[59] S. W. Golomb, and L.R. Welch, "Perfect Codes in the Lee Metric and the Packing of Polyominoes," *Siam J. Appl. Math,* pp. 302-317, Jan., 1970.

[60] A. Jiang, R. Mateescu, M. Schwartz, and J. Bruck, "Rank Modulation for Flash Memories," *IEEE Trans. Information Theory,* pp. 2659-2673, June, 2009.

[61] Q. Huang, S. Lin, and K. Abdel-Ghaffar, "Error-Correcting Codes for Flash Coding," *IEEE Information Theory and Applications Workshop (ITA),* pp. 1-23, Feb. 2011.

[62] I. Tamo, and M. Schwartz, "Correcting Limited-Magnitude Errors in the Rank-Modulation Scheme," *IEEE Trans. Information Theory*, pp.1-9, July 2009.

[63] F. Balasa, *Data Storage*, Vienna:In-Tech, 2010.

Vita

Nattakan Puttarak received a Bachelor Degree in Electronics and Telecommunications Engineering from the King Mongkut's University of Technology Thonburi (KMUTT), Bangkok, Thailand in 2003. She joined the graduate school of Lehigh University under the support from Thai Government scholarship to pursue the Doctor of Philosophy degree. She successfully got Master Degree in Electrical Engineering in 2007, and continued on with her Ph.D Degree at Lehigh. She joined Prof. Tiffany Jing Li's group since she first started her Master's thesis in 2005, and is expected to get her Ph.D degree in Electrical Engineering in August 2011.

Nattakan's research interests fall in the area of data storage, including both the mainstream systems of hard drives and the emerging technology of flash drives. She has specifically focused on coding techniques for storage systems. This includes designing new error correction coding strategies and decoding algorithms to combat disk failure and recover lost data for small-scale disk arrays as well as large-scale data centers, analyzing their performances, and identifying best practices. This also includes developing new source coding and labeling techniques for flash memories to minimize the number of reset operations and increase the lifespan of the device.

Nattakan will work for the King Mongkut's Institute of Technology Lardkrabang (KMITL), Bangkok, Thailand, as a lecturer. She wishes to apply all the knowledge and experience she gained from Lehigh to help lift up the academic and educational level in her country.