UNIVERSITY OF CALIFORNIA, SAN DIEGO

Coding for Flash Memories

A dissertation submitted in partial satisfaction of the
requirements for the degree
Doctor of Philosophy

in

Electrical Engineering
(Communication Theory and Systems)

by

Eitan Yaakobi

2011

The dissertation of Eitan Yaakobi is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

Co-Chair

Co-Chair

University of California, San Diego

2011

DEDICATION

Dedicated to my parents and to the memory of Prof. Jack K. Wolf.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

Many people have contributed for my work in the past five years. Words are not enough in order to express my gratitude and thank each one of them.

First of all, I would like to express my deepest gratitude for my three advisors, Prof. Paul H. Siegel, Prof. Alexander Vardy, and Prof. Jack K. Wolf. Interacting with three of the best researchers in the world and enjoying their guidance as mentors was a tremendous benefit throughout my time at UCSD. I gained from their experience, enjoyed their patience, care, and encouragement along the way. Especially, I want to thank them for their belief in me right from the beginning. They helped me to push and reach the best out of myself.

I was very fortunate to interact with more researchers during the PhD program. This collaboration paved the way for a fruitful and very enjoyable research. I would like to thank Prof. Steven Swanson for the opportunity to collaborate with him and his lab and expanding my research into new and fascinating directions. I thank him also for serving in my committee and I thank Prof. Alon Orlitsky and Prof. Ilya Dumer for their time and effort in serving in my committee as well. I would like to thank my former M.Sc. advisor, Prof. Tuvi Etzion. His guidance in my research work and all other aspects of my life is extremely valuable and helped me a lot. I would like to thank Prof. Shuki Bruck and Prof. Andrew Jiang. Collaborating with them and their colleagues was a great honor for me. I want to thank them for establishing this new research area of coding for flash memories, which opened the path for my Phd research. I would like to thank all my other colleagues: Adrian Caulfield, Amy Chen, Joel Coburn, Lara Dolecek, Ryan Gabrys, Laura Grupp, Scott Kayser, Jing Ma, Hessam Mahdavifar, and Robert Mateescu. This collaboration had a tremendous contribution for my research and I am very grateful for that.

I would like to thank the staff of the CMRR and the ECE department staff: Arline Allen, Ray Descoteaux, Betty Manoulian, Gennie Miranda, Jan Neumann, Robert Rome, Shana Slebioda, and Iris Villanueva. You provided me with the perfect workplace and support for any administrative help I needed and much more than that.

I thank the current and previous STAR members: Sharon Aviran, Aman Bhatia, Brian Butler, Amir Hadi Djahanshahi, Federica Garin, Lynn Greiner, Junsheng Han, Toshio Ito, Aravind Iyengar, Seyhan Karakulak, Scott Kayser, Zsigmond Nagy, Minghai Qin, Marco Papaleo, Ori Shental, Mohammad Hossein Taghavi, Ido Tal, Saeed Sharifi Tehrani, Han Wang, Hao Wang, Zheng Wu, and Xiaojie Zhang, the non-volatile systems laboratory members: Adrian Caulfield, Joel Coburn, Laura Grupp, Hung-Wei Tseng, and Michael Wei, as well as current and past ECE students: Craig Armstrong, Steve Cho, Lorenzo Coviello, Shay Har-Noy, Hessam Mahdavifar, Mehmet Parlak, Mattew Pugh, Nevena Rakuljic, Rohit Ramanujam, and Sheu-Sheu Tan. I have benefited from the interaction I had with each one of them whether for research, organizing seminars, and any other activity on campus.

A special thank for all my friends in San Diego, Israel, and other parts of the world. Their love, belief in me, and serving as my extended family made me feel home and gave me the strength along this trip. It is impossible to go through all their names but I want to thank each one of them individually for this tremendous help and support.

I want to thank my loving and caring family: my brother Arik and his wife Revital, their great kids Opal and Orian, my sister Hadar and her Fiance Roey, my sister Maayan and her husband Benny, and my wonderful parents Baruch and Carmela. I am grateful for their love, endless support and help in any aspect of my life which made this trip so much easier for me.

Last and most importantly, I would like to cherish the memory of Jack Wolf and dedicate him this work. His contribution for my work cannot be described in just words. The guidance I was fortunate to receive from Jack for my research and more importantly for life will be remembered with me for ever.

This thesis was supported in part by the University of California Lab Fees Research Program, Award No. 09-LR-06-118620-SIEP, the Intel Ph.d. Fellowship program, the ISEF Foundation, and the Center for Magnetic Recording Research at the University of California, San Diego.

Chapter 2 is in part a reprint of the material in the papers: E. Yaakobi, P.H. Siegel, and J.K. Wolf, "Buffer Codes for Multi-Level Flash Memory," *IEEE International Symposium on Information Theory*, Poster session, Toronto, Canada, July 2008, and H. Mahdavifar, P.H. Siegel, A. Vardy, J.K. Wolf, and E.Yaakobi, "A Nearly Optimal Construction of Flash Codes," *Proc. IEEE International Symposium on Information Theory*, pp. 1239–1243, Seoul, Korea, June 2009. Chapter 3 is in part a reprint of the material in the papers: E. Yaakobi, S. Kayser, P.H. Siegel, A. Vardy, and J.K. Wolf, "Efficient Two-Write WOM-Codes," *Proc. IEEE Information Theory Workshop*, Dublin, Ireland, August-September 2010, and S. Kayser, E. Yaakobi, P.H. Siegel, A. Vardy, and J.K. Wolf, "Multiple-Write WOM-Codes," *48-th Annual Allerton Conference on Communication, Control and Computing*, Monticello, IL, September 2010. Chapter 4 is in part a reprint of the material in the paper: E. Yaakobi, P.H. Siegel, A. Vardy, and J.K. Wolf, "Multiple Error-Correcting WOM-Codes," *Proc. IEEE International Symposium on Information Theory*, pp. 1933–1937, Austin, Texas, June 2010. Chapter 5 is in part a reprint of the material in the paper: E. Yaakobi, P.H. Siegel, A. Vardy, and J.K. Wolf, “On Codes that Correct Asymmetric Errors with Graded Multiple Distribution,” to appear *IEEE International Symposium on Information Theory*, Saint Petersburg, Russia, July 2011. Chapter 6 is in part a reprint of the material in the paper: E. Yaakobi, A. Jiang, P.H. Siegel, A. Vardy, and J.K. Wolf, “On The Parallel Programming of Flash Memory Cells,” *Proc. IEEE Information Theory Workshop*, Dublin, Ireland, August-September 2010. Chapter 7 is in part a reprint of the material in the paper: A. Jiang, R. Mateescu, E. Yaakobi, J. Bruck, P.H. Siegel, A. Vardy, and J.K. Wolf, “Storage Coding for Wear Leveling in Flash Memories,” *IEEE Transactions on Information Theory*, vol. 56, no. 10, pp. 5290–5299, October 2010. Chapter 8 is in part a reprint of the material in the paper: E. Yaakobi, J. Ma, L. Grupp, P.H. Siegel, S.Swanson, and J.K. Wolf, “Error Characterization and Coding Schemes for Flash Memories,” *Proc. Workshop on the Application of Communication Theory to Emerging Memory Technologies*, Miami, Florida, December 2010.

VITA

2005	B.A. in Computer Science, Technion - Israel Institute of Technology
2005	B.A. in Mathematics, Technion - Israel Institute of Technology
2007	M.Sc. in Computer Science, Technion - Israel Institute of Technology
2011	Ph.D. in Electrical and Computer Engineering, University of California, San Diego

PUBLICATIONS

Eitan Yaakobi, Paul H. Siegel, Alexander Vardy, and Jack K. Wolf, "Multiple Error-Correcting WOM-Codes," submitted to *IEEE Transactions on Information Theory*, May 2011.

Minghai Qin, Eitan Yaakobi, and Paul H. Siegel "Time-Space Constrained Codes for Pahse-Change Memories," submitted to *Globecom Communications Conference*, Houston, Texas, December 2011.

Eitan Yaakobi, Paul H. Siegel, Alexander Vardy, and Jack K. Wolf, "On Codes that Correct Asymmetric Errors with Graded Multiple Distribution," to appear *IEEE International Symposium on Information Theory*, Saint Petersburg, Russia, July 2011.

Eitan Yaakobi, Jing Ma, Laura Grupp, Paul H. Siegel, Steven Swanson, and Jack K. Wolf, "Error Characterization and Coding Schemes for Flash Memories," *Proc. Workshop on the Application of Communication Theory to Emerging Memory Technologies*, Miami, Florida, December 2010.

Anxiao (Andrew) Jiang, Robert Mateescu, Eitan Yaakobi, Jehoshua Bruck, Paul H. Siegel, Alexander Vardy, and Jack K. Wolf, "Storage Coding for Wear Leveling in Flash Memories," *IEEE Transactions on Information Theory*, vol. 56, no. 10, pp. 5290–5299, October 2010.

Scott Kayser, Eitan Yaakobi, Paul H. Siegel, Alexander Vardy, and Jack K. Wolf, "Multiple-Write WOM-Codes," *48-th Annual Allerton Conference on Communication, Control and Computing*, Monticello, IL, September 2010.

Tuvi Etzion, Alexander Vardy, and Eitan Yaakobi, "Dense Error-Correcting Codes in the Lee Metric," *Proc. IEEE Information Theory Workshop*, Dublin, Ireland, August-September 2010.

Eitan Yaakobi, Anxiao (Andrew) Jiang, Paul H. Siegel, Alexander Vardy, and Jack K. Wolf, "On The Parallel Programming of Flash Memory Cells," *Proc. IEEE Information Theory Workshop*, Dublin, Ireland, August-September 2010.

Eitan Yaakobi, Scott Kayser, Paul H. Siegel, Alexander Vardy, and Jack K. Wolf, "Efficient Two-Write WOM-Codes," *Proc. IEEE Information Theory Workshop*, Dublin, Ireland, August-September 2010.

Eitan Yaakobi and Tuvi Etzion, "High Dimensional Error-Correcting Codes," *Proc. IEEE International Symposium on Information Theory*, pp. 1178–1182, Austin, Texas, June 2010.

Eitan Yaakobi, Paul H. Siegel, Alexander Vardy, and Jack K. Wolf, "Multiple Error-Correcting WOM-Codes," *Proc. IEEE International Symposium on Information Theory*, pp. 1933–1937, Austin, Texas, June 2010.

Laura Grupp, Adrian Caulfield, Joel Coburn, Steven Swanson, Eitan Yaakobi, Paul Siegel, Jack Wolf, "Characterizing Flash Memory: Anomalies, Observations, and Applications," *MICRO 42*, New-York, New-York, December 2009.

Anxiao (Andrew) Jiang, Robert Mateescu, Eitan Yaakobi, Jehoshua Bruck, Paul H. Siegel, Alexander Vardy, and Jack K. Wolf, "Storage Coding for Wear Leveling in Flash Memories," *Proc. IEEE International Symposium on Information Theory*, pp. 1234–1238, Seoul, Korea, June 2009.

Hessam Mahdavifar, Paul H. Siegel, Alexander Vardy, Jack K. Wolf, and Eitan Yaakobi, "A Nearly Optimal Construction of Flash Codes," *Proc. IEEE International Symposium on Information Theory*, pp. 1239–1243, Seoul, Korea, June 2009.

Tuvi Etzion and Eitan Yaakobi, "Error-Correction of Multidimensional Bursts", *IEEE Transactions on Information Theory*, vol. 55, no. 3, pp. 961–976, March 2009.

Eitan Yaakobi, Alexander Vardy, Paul H. Siegel, and Jack K. Wolf, "Multidimensional Flash Codes," *Proc. 46-th Annual Allerton Conference on Communication, Control and Computing*, Monticello, IL, September 2008.

Eitan Yaakobi, Paul H. Siegel, and Jack K. Wolf, "Buffer Codes for Multi-Level Flash Memory," *IEEE International Symposium on Information Theory*, Poster session, Toronto, Canada, July 2008.

Eitan Yaakobi and Tuvi Etzion, "Error Correction of Multidimensional Bursts," *Proc. IEEE International Symposium on Information Theory*, pp. 1178–1182, Nice, France, June 2007.

ABSTRACT OF THE DISSERTATION

Coding for Flash Memories

by

Eitan Yaakobi

Doctor of Philosophy in Electrical Engineering
(Communication Theory and Systems)

University of California, San Diego, 2011

Professor Paul H. Siegel, Co-Chair
Professor Alexander Vardy, Co-Chair

Flash memories are, by far, the most important type of non-volatile memory in use today. They are employed widely in mobile, embedded, and mass-storage applications, and the growth in this sector continues at a staggering pace. Moreover, since flash memories do not suffer from the mechanical limitations of magnetic disk drives, solid-state drives have the potential to upstage the magnetic recording industry in the foreseeable future. The research goal of this dissertation is the discovery of new coding theory methods that supports efficient design of flash memories.

Flash memory is comprised of blocks of cells, wherein each cell can take on $q \geqslant 2$ levels. While increasing the cell level is easy, reducing its level can be accomplished only by erasing an entire block. Such block erasures are not only time-consuming, but also degrade the memory lifetime.

Our main contribution in this research is the design of rewriting codes that maximize the number of times that information can be written prior to incurring a block erasure. Examples of such coding schemes are *flash/floating codes* and *buffer codes*, introduced by Jiang and Bruck et al. in 2007, and *WOM-codes* that were presented by Rivest and Shamir almost three decades ago. The overall goal in these codes is to maximize the amount of information written to a fixed number of cells in a fixed number of writes.

Furthermore, the design of error-correcting codes in flash memories is extensively studied. It is shown how to modify WOM-codes to support an error-correction capability. Motivated by the asymmetry of the error behavior of flash memories and the work by Cassuto et al., a coding scheme to correct asymmetric errors is presented. An extensive empirical database of errors was used to develop a comprehensive understanding of the error behavior as well as to design specific error-correcting codes for flash memories.

This research on flash memories is expanded to other directions. *Wear leveling* techniques are widely used in flash memories in order to reduce and balance block erasures. It is shown that coding schemes to be used in these techniques can significantly reduce the number block erasures incurred during data movement. Also, the design of parallel cell programming algorithms is studied for the specific constraints and behavior of flash cells.

Chapter 1

Introduction

1.1 Background

Coding theory was effectively born in 1948, with the publication of Shannon's celebrated classic paper [3]. It was recognized early on that error-correction coding is a powerful system design technique that can fundamentally change the trade-offs in a communication system. Indeed, in order to improve performance, error-control coding has been widely used by the communications and data storage industries. In particular, the research in this dissertation focuses on the relevant problems arising from flash memory systems.

Any communication channel can be considered as a transmission of information either from one place to another — space domain, or from one time to another — time domain. In both types of communication channels, the information cannot be perfectly transmitted and a noisy version of it is received. The noise can be caused by various reasons. If the information is transmitted in the space domain then any natural source such as weather conditions, radiation, thermal effects, etc. can cause noise in the channel. In the time domain, the information is stored on a memory device and any physical defect or degradation of the memory reliability can damage the stored data. For example, a scratch on a CD will corrupt the bits that are stored in that area on the CD. Both channels use the same types of signal processing systems and error-control codes in order to transmit the information reliably.

Data storage devices rely upon error detection and correction (EDAC) codes to ensure highly reliable information retrieval. Optical storage devices, such as CD- and DVD-based recorders, allocate significant overhead for the redundancy introduced by the encoding of data into codewords. High-performance hard disk drives also devote overhead for high-rate EDAC codes that can correct multiple erroneous symbols within a codeword. The powerful codes used in these storage devices are the culmination of decades of research and development, and efforts to design more powerful and efficient EDAC coding algorithms are ongoing.

Non-volatile data storage devices, particularly those based upon flash memory technologies, are revolutionizing the way we access and manipulate information. They have many attractive features compared to magnetic hard disk drives, including their compactness, shock resistance, lack of moving parts, and lower data-access time. Flash memory is now the storage medium of choice in portable consumer electronic applications, and high performance solid-state drives (SSDs) are also being introduced into mobile computing, enterprise storage, data warehousing, and data-intensive computing systems. Accordingly, there is a surge in interest in the refinement, development, and expanded commercial use of these non-volatile memory technologies. On the other hand, these technologies present major challenges in the areas of device reliability, endurance, and energy efficiency. These challenges can be overcome, in part, through innovative coding and data handling techniques.

Flash memory chips may use single-level cell (SLC) technology, where each cell can store one binary digit, or multi-level cell (MLC) technology, where each cell can store multiple binary digits. First generation flash storage devices have used only low-redundancy EDAC codes that offer minimal error correction and detection capabilities, such as single-bit error-correcting Hamming codes and error-detecting cyclic redundancy check (CRC) codes. The demand for increased storage capacity, coupled with the introduction of MLC flash technology, has created the need for more powerful ECC methods, such as BCH codes and Low-Density Parity-Check (LDPC) codes.

1.2 Flash Memory Basics

A flash memory consists of an array of floating-gate cells, organized into blocks (a typical block comprises roughly 2^{20} cells). Hot-electron injection is used to inject electrons into a cell, where they become trapped. The Fowler-Nordheim tunneling mechanism (field emission) can be used to remove electrons from an entire block of cells, thereby discharging them. The level of a cell is a function of the amount of charge (electrons) trapped within it. Historically, flash cells have been designed to store only two values (one bit); however, multilevel flash cells are actively being developed and are already in use in many devices [3]. In multilevel flash cells, voltage is quantized to q discrete threshold values. The parameter q can range from $q = 2$ (the conventional two-state case) up to $q = 16$.

The most conspicuous property of flash storage is its inherent asymmetry between cell programming (charge placement) and cell erasing (charge removal). While adding charge to a single cell is a fast and simple operation, removing charge from a cell is very difficult. In fact, flash memories do not allow a single cell to be erased; rather, only entire blocks (comprising up to 2^{20} or more cells) can be erased. Thus, a single cell erase operation requires the cumbersome process of copying an entire block to a temporary location, erasing it, and then re-programming all the cells except one. Moreover, since over-programming (raising the charge of a cell above its intended level) can only be corrected by a block erasure, in practice, a conservative procedure is used for programming a cell. Charge is injected into the cell over numerous rounds; after every round, the charge level is measured and the next-round injection is configured, so that the charge gradually approaches its desired level. All this is extremely costly in time and energy. Such block erasures are not only time-consuming, but also degrade the lifetime of the memory. A typical block can tolerate about $10^4 - 10^5$ or fewer erasures.

1.3 Codes for Flash Memories

As flash memory has become ubiquitous in recent years, finding solutions for its limited lifetime and asymmetric programming behavior has become an important challenge. Examples of such coding solutions were given in [4,8,9,11,12,17,31]. The goal in these codes is to rewrite the information into flash memories while preserving the constraint that on each write cells can only increase their value. In fact, the model and behavior of a flash memory is very similar to the well-studied write-once memory (WOM) model. Inspired by memories such as punch cards and optical disks, Rivest and Shamir first introduced WOM-codes in 1982. Similarly to flash, in these memories, the media can be represented as a collection of write-once bit locations, each of which initially represents a bit value 0 that be irreversibly overwritten with a bit value 1. Constructions of WOM-codes were given for example in [2, 5, 5, 13, 17, 18] and were recently improved in [11,19]. Error correcting WOM-codes were also studied in [20,21] and more recent constructions appeared in [29].

Slightly different and yet very related are the rank modulation codes [12, 13]. In rank modulation, the information is not stored according to the exact cell levels but rather by the cell permutation which is derived from these levels. Other works explore efficient data movement in flash memories that minimize the number of block erasures as well as extra space [13, 17] and fast cell programming algorithms of flash memories [7, 8, 27].

1.4 Dissertation Overview

After understanding the model and constraints in flash memory we can overview the problems which this dissertation solves using coding theory tools.

In Chapter 2, we present rewriting codes for flash memories. The first part of this chapter focuses on the *flash codes* problem and a nearly optimal construction of such codes is presented. In the second part, *buffer codes* are studied. An upper bound and a construction of such codes are presented which improve upon the related previous work.

WOM-codes are studied in Chapter 3. We first show how to construct two-write WOM-codes that improve upon the best previously known. These codes are also used to construct codes for the Blackwell channel. Then, we show how to extend these codes to an arbitrary number of writes, giving the best known WOM-codes for $3 \leqslant t \leqslant 10$ writes.

The study of WOM-codes is continued in Chapter 4. Here, error-correcting WOM-codes are constructed which improve upon the previously known single-error-correcting WOM-codes. Extensions for double- and triple-error-correcting WOM-codes are given as well. For triple-error-correction, we introduce the notion of strong cyclic error-correcting codes. Finally, we show how to design WOM-codes for the correction of any arbitrary number of errors.

Chapter 5 describes a new error model for multi-level flash memories based upon a graded distribution of asymmetric errors of limited magnitudes. We show how to use a previous construction by Cassuto et al. [5] of asymmetric limited-magnitude error-correcting codes in order to develop a family of codes that correct asymmetric errors in the new model.

Chapter 6 presents algorithms for programming flash memory cells. First, a model which describes the programming of flash memory cells is given. According to this model, algorithms to program the cells in parallel are given. Modifications to programming with noise and inter-cell interference are given as well. Then, we show an algorithm that is used to get information about the characteristics of the cells and using this information we show how to program a cell with feedback.

Chapter 7 studies how to efficiently move data in flash memories. We show how to utilize coding schemes in order to construct codes that reduce the number of auxiliary blocks used in the algorithm as well as the number of block erasures.

Finally, Chapter 8 uses empirical data to characterize the flash error behavior. This information will be useful in the design of new error-correcting codes for flash memories. Furthermore, we explore in this chapter the implementation of WOM-codes in flash memories.

Bibliography

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

[2] Y. Cassuto, M. Schwartz, V. Bohossian, and J. Bruck, “Codes for asymmetric limited-magnitude errors with application to multi-level flash memories,” *IEEE Trans. Inform. Theory*, vol. 56, no. 4, pp. 1582–1595, April 2010.

[3] C.E. Shannon, “A mathematical theory of communication,” *Bell Systems Tech. J.*, vol. 27, pp. 379–423 and pp. 623–656, 1948.

[4] F. Chierichetti, H. Finucane, Z. Liu, and M. Mitzenmacher, “Designing floating codes for expected performance,” to appear in *IEEE Trans. Inform. Theory*.

[5] G.D. Cohen, P. Godlewski, and F. Merkx, “Linear binary code for write-once memories,” *IEEE Trans. Inform. Theory*, vol. 32, no. 5, pp. 697–700, October 1986.

[6] T. Etzion, A. Vardy, and E. Yaakobi, “Dense error-correcting codes in the Lee metric,” *Proc. IEEE Inform. Theory Workshop*, Dublin, Ireland, August-September 2010.

[7] T. Etzion and E. Yaakobi, “Error-correction of multidimensional bursts”, *IEEE Trans. Inform. Theory*, vol. 55, no. 3, pp. 961–976, March 2009.

[8] P. Godlewski, “WOM-codes construits à partir des codes de Hamming,” *Discrete Math.*, no. 65, no. 3, pp. 237–243, July 1987.

[9] L. Grupp, A. Caulfield, J. Coburn, S. Swanson, E. Yaakobi, P.H. Siegel, and J.K. Wolf, “Characterizing flash memory: anomalies, observations, and applications,” *MICRO-42*, pp. 24–33, December 2009.

[10] A. Jiang, V. Bohossian, and J. Bruck, “Floating codes for joint information storage in write asymmetric memories,” *Proc. IEEE Int. Symp. Inform. Theory* pp. 1166–1170, Nice, France, June 2007.

[11] A. Jiang and J. Bruck, “Joint coding for flash memory storage,” *Proc. IEEE Int. Symp. Inform. Theory*, pp. 1741–1745, Toronto, Canada, July 2008.

[12] A. Jiang and J. Bruck, “On the capacity of flash memories,” *Proc. Int. Symp. on Inform. Theory and Its Applications*, pp. 94–99, Auckland, New Zealand, 2008.

[13] A. Jiang, M. Langberg, R. Mateescu, and J. Bruck, “Data movement and aggregation in flash memories,” *Proc. IEEE Int. Symp. Inform. Theory*, pp. 1918–1922, Austin, Texas, June 2010.

[14] A. Jiang, M. Landberg, M. Schwartz, and J. Bruck, “Universal rewriting in constrained memories,” *Proc. IEEE Int. Symp. Inform. Theory*, pp. 1219–1223, Seoul, Korea, July 2009.

[15] A. Jiang and H. Li, “Optimized cell programming for flash memories,” *Proc. IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM)*, pp. 914–919, Victoria, BC, August 2009.

[16] A. Jiang, R. Mateescu, M. Schwartz, and J. Bruck, "Rank modulation for flash memories," *IEEE Trans. Inform. Theory*, vol. 55, no. 6, pp. 2659–2673, October 2010.

[17] A. Jiang, R. Mateescu, E. Yaakobi, J. Bruck, P.H. Siegel, A. Vardy, and J.K. Wolf, "Storage coding for wear leveling in flash memories," *IEEE Transactions on Information Theory*, vol. 56, no. 10, pp. 5290–5299, October 2010.

[18] A. Jiang, M. Schwartz, and J. Bruck, "Correcting charge-constrained errors in the rank modulation scheme," *IEEE Trans. Inform. Theory*, vol. 56, no. 5, pp. 2112–2120, May 2010.

[19] S. Kayser, E. Yaakobi, P.H. Siegel, A. Vardy, and J.K. Wolf, "Multiple-write WOM-codes," *Proc. 48-th Annual Allerton Conference on Communication, Control and Computing*, Monticello, IL, September 2010.

[20] H. Mahdavifar, P.H. Siegel, A. Vardy, J.K. Wolf, and E. Yaakobi, "A nearly optimal construction of flash codes," *Proc. IEEE Int. Symp. Inform. Theory*, pp. 1239–1243, Seoul, Korea, July 2009.

[21] F. Merkx, "Womcodes constructed with projective geometries," *Traitement du signal*, vol. 1, no. 2-2, pp. 227–231, 1984.

[22] R.L. Rivest and A. Shamir, "How to reuse a write-once memory," *Inform. and Contr.*, vol. 55, no. 1–3, pp. 1–19, December 1982.

[23] Y. Wu, "Low complexity codes for writing write-once memory twice," *Proc. IEEE Int. Symp. Inform. Theory*, pp. 1928–1932, Austin, Texas, June 2010.

[24] Y. Wu and A. Jiang, "Position modulation code for rewriting write-once memories," accepted by *IEEE Trans. Inform. Theory*, October 2010.

[25] E. Yaakobi and T. Etzion, "Error correction of multidimensional bursts," *Proc. IEEE Int. Symp. Inform. Theory*, pp. 1381–1385, Nice, France, June 2007.

[26] E. Yaakobi and T. Etzion, "High dimensional error-correcting codes," *Proc. IEEE Int. Symp. Inform. Theory*, pp. 1178–1182, Austin, Texas, June 2010.

[27] E. Yaakobi, A. Jiang, P.H. Siegel, A. Vardy, and J.K. Wolf, "On the parallel programming of flash memory cells," *Proc. IEEE Inform. Theory Workshop*, Dublin, Ireland, August-September 2010.

[28] E. Yaakobi, S. Kayser, P.H. Siegel, A. Vardy, and J.K. Wolf, "Efficient two-write WOM-codes," *Proc. IEEE Inform. Theory Workshop*, Dublin, Ireland, August-September 2010.

[29] E. Yaakobi, J. Ma, L. Grupp, P.H. Siegel, S. Swanson, and J.K. Wolf, "Error characterization and coding schemes for flash memories," *Proc. Workshop on the Application of Communication Theory to Emerging Memory Technologies*, Miami, Florida, December 2010.

[30] E. Yaakobi, P.H. Siegel, A. Vardy, and J.K. Wolf, "Multiple error-correcting WOM-codes," *Proc. IEEE Int. Symp. Inform. Theory*, pp. 1933–1937, Austin, Texas, June 2010.

[31] E. Yaakobi, P.H. Siegel, and J.K. Wolf, “Buffer codes for multi-level flash memory,” *Proc. IEEE Int. Symp. Inform. Theory*, Poster session, Toronto, Canada, July 2008.

[32] E. Yaakobi, A. Vardy, P.H. Siegel, and J.K. Wolf, “Multidimensional flash codes,” *Proc. 46-th Allerton Conference on Communication, Control and Computing*, Monticello, IL, September 2008.

[33] G. Zémor, “Problèmes combinatoires liés à l’écriture sur des mémoires,” Ph.D. Dissertation, ENST, Paris, France, November 1989.

[34] G. Zémor and G.D. Cohen, “Error-correcting WOM-codes,” *IEEE Trans. Inform. Theory*, vol. 37, no. 3, pp. 730–734, May 1991.

Chapter 2

Rewriting Codes for Flash Memories

2.1 Introduction

In Chapter 1, we described how flash memories work and in particular the constraints of such memories. The most conspicuous property of flash-storage technology is its inherent asymmetry between cell programming (charge placement) and cell erasing (charge removal). In fact, flash technology does not allow a single cell to be erased — rather, only entire blocks can be erased. This constraint makes it important to design coding schemes that maximize the number of times information stored in a flash memory can be written (and re-written) prior to incurring a block erasure.

Such coding schemes — known as ***floating codes*** or ***flash codes*** and ***buffer codes*** — were first introduced in [1,7,8] a few years ago. Since then, several more papers on this subject have appeared in the literature [5,9,9,11,12,17]. It should be pointed out that flash codes and buffer codes can be regarded as examples of memories with constrained source, which were described in [11]. Yet another example of such codes are the write-once memory (WOM) codes [2,3,5], that have been studied since the early 1980s. In fact, flash codes may be regarded as a generalization of WOM-codes. Slightly different and yet very related are the rank modulation codes [12,13]. In rank modulation, the information is not stored according to the exact cell levels but rather by the cell permutation which is derived from these levels.

2.2 Preliminaries and Flash Codes Definition

Let us first give a precise definition of flash codes that were introduced less formally in the previous section. We use $\{0,1\}^k$ to denote the set of binary vectors of length k, and refer to the elements of this set as ***information vectors***. The set of possible levels for each cell is denoted by $\mathcal{A}_q = \{0,1,\ldots,q-1\}$ and thought of as a subset of the integers. The q^n vectors of length n over $\mathcal{A}_q$ are called ***cell-state vectors***. With this notation, any flash code $\mathbb{C}$ can be specified in terms of two functions: an encoding map $\mathcal{E}$ and a decoding map $\mathcal{D}$. The ***decoding map*** $\mathcal{D}\colon \mathcal{A}_q^n \to \{0,1\}^k$ indicates for each cell-state vector $\boldsymbol{x} \in \mathcal{A}_q^n$ the corresponding information vector. In turn, the ***encoding map*** $\mathcal{E}\colon \{0,1,\ldots,k-1\} \times \mathcal{A}_q^n \to \mathcal{A}_q^n \cup \{\mathsf{E}\}$ assigns to every index i and cell-state vector $\boldsymbol{x} \in \mathcal{A}_q^n$, another cell-state vector $\boldsymbol{y} = \mathcal{E}(i,\boldsymbol{x})$ such that $y_j \geqslant x_j$ for all j and $\mathcal{D}(\boldsymbol{y})$ differs from $\mathcal{D}(\boldsymbol{x})$ *only* in the i-th position. If no such $\boldsymbol{y} \in \mathcal{A}_q^n$ exists, then $\mathcal{E}(i,\boldsymbol{x}) = \mathsf{E}$ indicating that block erasure is required. To bootstrap the encoding process, we assume that the initial state of the n memory cells is $(0,0,\ldots,0)$. Henceforth, iteratively applying the encoding map, we can determine how *any sequence* of transitions $0 \to 1$ or $1 \to 0$ in the k information bits maps into a sequence of cell-state vectors, eventually terminated by the block erasure. This leads to the following definition.

Definition 2.2.1. *An $(n,k)_q$ flash code $\mathbb{C}(\mathcal{D},\mathcal{E})$* ***guarantees** t **writes*** *if for all sequences of up to t transitions $0 \to 1$ or $1 \to 0$ in the k information bits, the encoding map $\mathcal{E}$ does not produce the block erasure symbol E. If so, we say that $\mathbb{C}$ is an $(n,k,t)_q$ code, and define the* ***deficiency of*** $\mathbb{C}$ *as $\delta(\mathbb{C}) = n(q-1) - t$.*

In addition to this definition, we will also use the following terminology. Given a vector $\boldsymbol{x} = (x_1,x_2,\ldots,x_m)$ over $\mathcal{A}_q$, we define its ***weight*** as $\mathrm{wt}(\boldsymbol{x}) = x_1 + x_1 + \cdots + x_m$ (where the addition is over the integers), and its ***parity*** as $\mathrm{wt}(\boldsymbol{x}) \bmod 2$.

2.3 Two Bits Flash Codes

In this section, we present our first construction of flash codes. In [8], a construction for storing two bits is presented and is shown to be optimal. The construction given here will be proved to be optimal as well and we believe that it is more intuitive.

In this construction, the leftmost and rightmost cells correspond to the first and second bit, respectively. When rewriting, if the first or second bit changes its value then the leftmost or rightmost cell of level less than $q-1$ is increased by one level, respectively. In general, the cell-state vector has the following form:

$$(q-1,\ldots,q-1,x_i,0,\ldots,0,x_j,q-1,\ldots,q-1),$$

where $0 < x_i, x_j \leqslant q-1$. This principle repeats itself until only one cell is left with level less than $q-1$. Then, this cell is used to store two bits according to its residue modulo 4. If this residue is $0,1,2,3$ then the value of the bits is $(v_1,v_2) = (0,0),(1,0),(0,1),(1,1)$, respectively. The construction is presented for odd values of q and we will discuss later how to modify it for even values as well. In what follows, these maps are described algorithmically, using (C-like) pseudo-code notation.

Decoding map $\mathcal{D}_2$: The input to this map is a cell-state vector $\boldsymbol{x} = (x_1,x_2,\ldots,x_n)$. The output is the corresponding two-bit information vector (v_1,v_2).

```
$i_1$ = find_left_cell($y_1,y_2,\ldots,y_n$);
$i_2$ = find_right_cell($y_1,y_2,\ldots,y_n$);
if($i_2$ == 0)   // all cells are full
{ $v_1$ = $q-1$ (mod 2); $v_2$ = $\lfloor$(($q-1$) (mod 4))/2$\rfloor$; }
if ($i_1$ == $i_2$)  // there is only one non-full cell
{ $v_1$ = $y_{i_1}$ (mod 2); $v_2$ = $\lfloor$($y_{i_1}$ (mod 4))/2$\rfloor$; }
if ($i_1$ != $i_2$)  // there are at least two non-full cells
{ $v_1$ = $y_{i_1}$ (mod 2); $v_2$ = $y_{i_2}$ (mod 2); }
```

Encoding map $\mathcal{E}_2$: The input to this map is a cell-state vector $\boldsymbol{x} = (x_1,x_2,\ldots,x_n)$, and an index $j \in \{1,2\}$ of the bit that has changed. Its output is either a new cell-state vector $\boldsymbol{y} = (y_1,y_2,\ldots,y_n)$ or the erasure symbol E.

```
($y_1,y_2,\ldots,y_n$) = ($x_1,x_2,\ldots,x_n$);
$i_1$ = find_left_cell($y_1,y_2,\ldots,y_n$);
$i_2$ = find_right_cell($y_1,y_2,\ldots,y_n$);
if($i_2$ == 0) return $\mathsf{E}$;
if ($i_1$ == $i_2$)  // there is only one non-full cell
{ if($j$ == 2) $a$ = 2;
  else $a$ = $j$ + $2\cdot$($y_{i_1}$ (mod 2));
  if($y_{i_1}$ + $a$ > $q-1$) return $\mathsf{E}$;
  else { $y_{i_1}$ = $y_{i_1}$ + $a$; return; } }
$y_{i_j}$ = $y_{i_j}$ + 1;
if (($i_2 - i_1$ == 1) $\wedge$ ($y_{i_j}$ == $q-1$))
{ $v_{i_j}$ = 0; $v_{i_{3-j}}$ = $y_{i_{3-j}}$ (mod 2);
  $a$ = $2\cdot v_2$ + $v_1$ − ($y_{i_{3-j}}$ (mod 4));
  if($a$ < 0)  $y_{i_{3-j}}$ = $y_{i_{3-j}}$ + 4 + $x$;
  else  $y_{i_{3-j}}$ = $y_{i_{3-j}}$ + $a$; }
```

The function **`find_left_cell(`**$y_1, y_2, \ldots, y_n$**`)`** finds the leftmost cell of level less than $q-1$ and if there is not such a cell then it returns $n+1$. Similarly, the function **`find_right_cell(`**$y_1, y_2, \ldots, y_n$**`)`** finds the rightmost cell of level less than $q-1$ and if there is not such a cell then it returns 0. The notation y_{i_j} stands for the variable y_{i_1} in case $j=1$, and y_{i_2} if $j=2$. The same rule applies for $y_{i_{3-j}}$. The symbol $\wedge$ stands for the logical operator "and". The next theorem proves the number of writes this construction guarantees.

Theorem 2.3.1. *If there are n q-level cells and q is odd, then the code $\mathbb{C}(\mathcal{D}_2, \mathcal{E}_2)$ guarantees at least $t = (n-1)(q-1) + \left\lfloor \frac{q-1}{2} \right\rfloor$ writes before erasing.*

Proof. As long as there is more than one cell of level less than $q-1$, the weight of the cell-state vector increases by one on each write. This may change only after at least $(n-1)(q-1)$ writes. Assume that there is only one cell of level less than $q-1$ after $s = (n-1)(q-1)+k$ writes, where $k \geqslant 0$, and call it the i-th cell. Starting this write, the different residues modulo 4 of the i-th cell correspond to the four possible two-bit information vector (v_1, v_2). Therefore, on the s-th write, we also need to increase the level of the i-th cell so it will correspond to the correct information vector on this write. For all succeeding writes, if the second bit changes then the i-th cell increases by two levels. If the first bit changes from 0 to 1 then the i-th cell increases by one level and otherwise by three levels. Therefore, if there are m more writes and $v_1 = 0$ then the i-th cell increases by at most $2m$ levels, and if there are m more writes and $v_1 = 1$ then the i-th cell increases by at most $2m+1$ levels.

Let us consider all possible values of k and the information vector (v_1, v_2) on the s-th write in order to calculate the number of guaranteed writes before erasing. Note that on the s-th write $(v_1 + v_2) \equiv s(\text{mod } 2)$. Furthermore, since q is odd, the value of the bit that is written changes from one to zero because it reaches level $q-1$, and thus the other bit has value $k(\text{mod } 2)$.

1. Assume $k(\text{mod } 4) = 0$, then $(v_1, v_2) = (0,0)$ and the level of the i-th cell does not increase on the s-th write. Since $v_1 = 0$, after m writes the cell increases by at most $2m$ levels. Hence, there are at least $\frac{q-1-k}{2}$ more writes and the total number of writes is at least
$$(n-1)(q-1) + k + \frac{q-1-k}{2} \geqslant (n-1)(q-1) + \frac{q-1}{2}.$$
2. Assume $k(\text{mod } 4) = 1$, then $(v_1, v_2) = (1,0)$ or $(v_1, v_2) = (0,1)$. If $(v_1, v_2) = (1,0)$ then on the s-th write the i-th cell does not increase its level and after m writes its level increases by at most $2m+1$ levels. If $(v_1, v_2) = (0,1)$ then the i-th cell increases by

one level and after m writes its level increases by at most $2m$ more levels. Hence, in both cases there are at least $\frac{q-2-k}{2}$ more writes. Together we get that the total number of writes is at least

$$(n-1)(q-1)+k+\frac{q-2-k}{2} \geqslant (n-1)(q-1)+\frac{q-1}{2}.$$

3. Assume $k(\bmod 4)=2$, then $(v_1,v_2)=(0,0)$ and the i-th cell increases by two levels on s-th write. Since $v_1=0$ after m more writes the cell increases by at most $2m$ levels and hence there are at least $\lfloor (q-1-(k+2))/2 \rfloor$ more writes, where $k \geqslant 2$. There total number of write is at least

$$(n-1)(q-1)+k+\frac{q-3-k}{2} \geqslant (n-1)(q-1)+\frac{q-1}{2}.$$

4. Assume $k(\bmod 4)=3$, then $(v_1,v_2)=(1,0)$ or $(v_1,v_2)=(0,1)$. If $(v_1,v_2)=(1,0)$ then on the s-th write the i-th cell increases by two levels and after m more writes it increases by at most $2m+1$ levels. If $(v_1,v_2)=(0,1)$ then the i-th cell increases by three levels and after m more writes it increases by at most $2m$ more levels. Hence there are at least $\frac{q-4-k}{2}$ more writes, where $k \geqslant 3$. Thus, the total number of writes is at least

$$(n-1)(q-1)+k+\frac{q-4-k}{2} \geqslant (n-1)(q-1)+\frac{q-1}{2}.$$

In any case, the guaranteed number of writes is $(n-1)(q-1)+\left\lfloor \frac{q-1}{2} \right\rfloor$. ■

For even values of q, the construction is very similar. As long as there is more than one cell of level less $q-1$ we follow the same rules for the encoding. For the decoding, since $q-1$ is no longer even, the value of v_1 is the parity of the cells $1,\ldots,i_1$, where i_1 is the leftmost cell of value less $q-1$. The value of v_2 is the parity of the cells $i_2, i_2+1,\ldots,n$, where i_2 is the rightmost cell of value less $q-1$. If there is only one left cell, then it represents a value of two bits as before according to its residue modulo 4. If the the index of the last available cell is i then

$$\begin{aligned} v_1 &= (i-1+y_i)(\bmod 2), \\ v_2 &= ((n-i)+\lfloor (y_i(\bmod 4))/2 \rfloor)(\bmod 2). \end{aligned}$$

Also, the last cell does not reach level $q-1$ so is always possible to distinguish what the last cell is. We omit the tedious details as the proof is similar to the case where q is odd.

2.4 Index-less Indexed Flash Codes

What is the smallest possible write deficiency $\delta_q(n,k)$ for an $(n,k,t)_q$ flash code, and how does it behave asymptotically as the code parameters k and n get large? The best-known lower bound, due to Jiang, Bohossian, and Bruck [8], asserts that

$$\delta_q(n,k) \geqslant \frac{1}{2}(q-1)\min\{n,k-1\}. \tag{2.1}$$

How closely can this bound be approached by code constructions? It appears that the answer to this question depends on the relationship between k and n. In this section, we are concerned mainly with the case where both k and n are large, and n is much larger than k (in particular, $n \geqslant k^2$). In Section 2.6, we will consider the case $k/n = \text{const}$. At the other end of the spectrum, the case $k > n$ has been studied in [11].

The first construction of flash codes for large k are the so-called *indexed flash codes*, due to Jiang and Bruck [9, 9]. In this construction, the k information bits are partitioned into $m_1 = k/k'$ subsets of k' bits each (with $k' \leqslant 6$) while the memory cells are subdivided into $m_2 \geqslant m_1$ groups of n' cells each. Additional memory cells (called ***index cells***) are set aside to indicate for each subset of k' bits which group of n' memory cells is used to store them. The deficiency of the resulting flash codes is $O(\sqrt{qn})$. Note that for $n \geqslant k$, the lower bound on write deficiency in (2.1) behaves as $\Omega(qk)$, and thus does not depend on n. Consequently, the gap between the Jiang-Bruck construction [9] and the lower bound could be arbitrarily large, especially when n is much larger than k.

In [17], a different construction of flash codes was proposed. These codes are based upon representing the n memory cells as a high-dimensional array, and achieve a write deficiency of $O(qk^2)$. Crucially, the deficiency of these codes does *not* depend on n. Nevertheless, there was still a significant gap between $O(qk^2)$ — which was the best currently known result — and the lower bound of $\Omega(qk)$.

Our point of departure are the indexed flash codes by Jiang and Bruck [9, 9], that were briefly described above. In this section, we eliminate the need for index cells — and, thus, the overhead associated with these cells — in the Jiang-Bruck construction [9]. This is achieved by "encoding" the indices into the order in which the cell levels are increased.

As in [9], we partition the n memory cells into m groups of n' cells each. However, while in [9] the value of n' is more or less arbitrary, in our construction $n' = k$. We henceforth refer to such groups of $n' = k$ cells as ***blocks*** (though they are not related to the *physical blocks* of floating-gate cells which comprise the flash memory). We will furthermore use, throughout this chapter, the following terminology. We say that:

- ▶ a block is ***full*** if all its cells are at level $q-1$;
- ▶ a block is ***empty*** if all its cells are at level zero;
- ▶ a block is ***active*** if it is neither full nor empty.

In our construction, each block represents *exactly one bit*. This implies that the total number of blocks, given by $m = \lfloor n/k \rfloor$, must be at least k, which in turn implies $n \geqslant k^2$. If n is not divisible by k, the remaining cells are simply left unused. Finally, we also assume that either k is even or q is odd. If this is not the case, we can invoke the same construction with k replaced by $k+1$ (and the last bit permanently set to zero).

The key idea is that each block is used to encode not only the current value of the bit that it represents, but also *which* of the k bits it represents. The value of the bit is simply the parity of the block. The index of the bit is encoded in the *order* in which the levels of the k cells are increased. For example, if the block stores the i-th bit, first the level of the i-th cell in the block is increased from 0 to $q-1$ in response to the transitions $0 \to 1$ and $1 \to 0$ in the bit value. Then, the same procedure is applied to the $(i+1)$-st cell, the $(i+2)$-nd cell, and so on, with the indices $i+1, i+2, \ldots$ interpreted cyclically (modulo k). This process is illustrated in the following example.

Example 2.4.1. Suppose that $k = 4$ and $q = 3$. If a block represents the first bit, then its cell levels will transition from $(0,0,0,0)$ to $(2,2,2,2)$ in the following order:

$$(0000) \to (1000) \to (2000) \to (2100) \to (2200)$$

$$\to (2210) \to (2220) \to (2221) \to (2222)$$

On the other hand, for a block that represents the second bit, the corresponding cell-writing order is given by:

$$(0000) \to (0100) \to (0200) \to (0210) \to (0220)$$

$$\to (0221) \to (0222) \to (1222) \to (2222)$$

The cell-writing orders for blocks that represent the third and fourth bits are given, respectively, by

$$(0000) \to (0010) \to (0020) \to (0021) \to (0022)$$

$$\to (1022) \to (2022) \to (2122) \to (2222)$$

and

$$(0000) \to (0001) \to (0002) \to (1002) \to (2002)$$
$$\to (2102) \to (2202) \to (2212) \to (2222)$$

Note that, unless a block is full, it is always possible to determine which cell was written first and, consequently, which of the $k = 4$ bits this block represents.

We now provide a precise specification of an $(n,k)_q$ flash code $\mathbb{C}$ based upon this idea, in terms of a decoding map $\mathcal{D}_A$ and an encoding map $\mathcal{E}_A$.

Decoding map $\mathcal{D}_A$: The input to this map is a cell-state vector $\boldsymbol{x} = (\boldsymbol{x}_1|\boldsymbol{x}_2|\cdots|\boldsymbol{x}_m)$, partitioned into m blocks. The output is the corresponding information vector $(v_0, v_1, \ldots, v_{k-1})$.

```
$(v_0, v_1, \ldots, v_{k-1})$ = $(0, 0, \ldots, 0)$;
for ($j$ = 1; $j \leqslant m$; $j$ = $j + 1$)
    if (active($\boldsymbol{x}_j$))
        { $i$ = read_index($\boldsymbol{x}_j$); $v_i$ = parity($\boldsymbol{x}_j$); }
```

Encoding map $\mathcal{E}_A$: The input to this map is a cell-state vector $\boldsymbol{x} = (\boldsymbol{x}_1|\boldsymbol{x}_2|\cdots|\boldsymbol{x}_m)$, partitioned into m blocks of k cells, and an index i of the bit that has changed. Its output is either a cell-state vector $\boldsymbol{y} = (\boldsymbol{y}_1|\boldsymbol{y}_2|\cdots|\boldsymbol{y}_m)$ or the erasure symbol E.

```
$(\boldsymbol{y}_1|\boldsymbol{y}_2|\cdots|\boldsymbol{y}_m)$ = $(\boldsymbol{x}_1|\boldsymbol{x}_2|\cdots|\boldsymbol{x}_m)$;
for ($j$ = 1; $j \leqslant m$; $j$ = $j + 1$)
    if (active($\boldsymbol{x}_j$) $\wedge$ (read_index($\boldsymbol{x}_j$) == $i$))
        { write($\boldsymbol{y}_j$); break; }
if ($j$ == $m + 1$)   // active block not found
    for ($j$ = 1; $j \leqslant m$; $j$ = $j + 1$)
        if (empty($\boldsymbol{x}_j$)) { write_new($i, \boldsymbol{y}_j$); break;}
if ($j$ == $m + 1$)   // no empty blocks remain
    return E;
```

To complete the specification of the flash code $\mathbb{C}(\mathcal{D}_A, \mathcal{E}_A)$, let us elaborate upon all the functions used in the pseudo-code above. The functions **active($\boldsymbol{x}$)**, respectively **empty($\boldsymbol{x}$)**, simply determine if the given block is active, respectively empty. The function **parity($\boldsymbol{x}$)** computes the parity of $\boldsymbol{x}$, defined in Section 5.2. Note that the parity of a full block is always zero (since $k(q-1)$ is even, by assumption). The function **read_index($\boldsymbol{x}$)** computes the bit-index encoded in an active block $\boldsymbol{x} = (x_0, x_1, \ldots, x_{k-1})$. This can be done as follows. Find all the zero cells in $\boldsymbol{x}$. Note that these cells always form one cyclically contiguous run, say $x_j, x_{j+1}, \ldots, x_{j+r}$ (where the indices are modulo k). Then the index of the corresponding bit is

$i = j + r + 1 \mod k$. If there are no zeros in $\boldsymbol{x}$, there must be exactly one cell, say x_j, whose level is strictly less than $q-1$. In this case, the bit-index is $i = j + 1 \mod k$. The function **`write(`y`)`** proceeds along similar lines. Find the single cyclically contiguous run of zeros in $(y_0, y_1, \ldots, y_{k-1})$, say $y_j, y_{j+1}, \ldots, y_{j+r}$. If $y_{j-1} < q-1$, increase y_{j-1} by one; otherwise set $y_j = 1$. If there are no zeros in $\boldsymbol{y}$, find the unique cell y_j such that $y_j < q-1$ and increase its level by one. Fin- ally, the function **`write_new(`i, y`)`** simply sets $y_i = 1$.

Theorem 2.4.1. *The write deficiency of the flash code $\mathbb{C}(\mathcal{D}_A, \mathcal{E}_A)$ described above is at most*

$$(k-1)\Big((k+1)(q-1) \; - \; 1\Big) \;=\; O(qk^2) \tag{2.2}$$

Proof. Note that at each instance, at most k of the m blocks are active. The encoding map $\mathcal{E}_A(i, x)$ produces the erasure symbol E when there are no more empty blocks, and the block representing the i-th bit is full. In the worst case, this may occur when there are $k-1$ active blocks, each using just one cell level. This contributes $(k-1)\big(k(q-1) - 1\big)$ unused cell levels. In addition, there are at most $k-1$ cells that are unused due to the partition into $m = \lfloor n/k \rfloor$ blocks of exactly k cells. These contribute at most $(k-1)(q-1)$ unused cell levels. ■

2.5 Nearly Optimal Construction

It is apparent from the proof of Theorem 2.4.1 that the deficiency of the flash code $\mathbb{C}(\mathcal{D}_0, \mathcal{E}_0)$, constructed in Section 2.4, is due primarily to the following: when writing stops, there are still potentially numerous unused cell levels. The key idea developed in this section is to *continue writing* after the encoding map $\mathcal{E}_0$ produces the erasure symbol E, utilizing those cell levels that are left unused by $\mathcal{E}_0$. Obviously, it is *not* possible to continue writing using the same encoding and decoding maps. However, it may be possible to do so if, at the point when $\mathcal{E}_0$ produces the erasure symbol E, we switch to a *different encoding procedure*, say $\mathcal{E}_1$. In fact, this idea can be applied iteratively: once $\mathcal{E}_1$ reaches its limit, we will transition to another encoding map $\mathcal{E}_2$, then yet another map $\mathcal{E}_3$, and so on.

Assuming that $k \equiv 0 \pmod 4$, here is one way to continue writing after the encoding map $\mathcal{E}_0$ has been exhausted. When $\mathcal{E}_0$ produces the erasure symbol E, we say that the *first stage* of encoding is over and transition to the *second stage*, as follows. First, we re-examine the cell-state vector $\boldsymbol{x} = (\boldsymbol{x}_1|\boldsymbol{x}_2|\cdots|\boldsymbol{x}_m)$ and re-partition it into $2m = 2\lfloor n/k \rfloor$ blocks of $k/2$ cells each. Most of these smaller blocks will be already full, but we may find some m_1 of them that are either empty or active (live). Observe that $m_1 \leqslant 2(k-1)$ since at the end of the first stage, there are at most $k-1$ active blocks of k cells, and each of them produces at most two live (non-full) blocks of $k/2$ cells.

If $m_1 \geqslant k$, we can continue writing as follows. Once again, each of the m_1 blocks will represent exactly one bit; as before, the value of this bit is determined by the parity of the block. As part of the transition from the first stage to the second stage, we record the current information vector $(v_0, v_1, \ldots, v_{k-1})$ in the first k of the m_1 live blocks, say $\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_k$. To this end, whenever $\texttt{parity}(\boldsymbol{x}_i) \neq v_{i-1}$, we increase the level of one of the cells in $\boldsymbol{x}_i$ by one; otherwise, we leave $\boldsymbol{x}_i$ as is.

Since the blocks now have $k/2$ cells rather than k cells, it is no longer possible to encode in each block *which* of the k information bits it represents. Therefore, we set aside for this purpose $2(k-1)\lceil\log_q(k+2)\rceil$ *index cells* (that are not used during the first stage). These cells are partitioned into $2(k-1)$ blocks of $\mu = \lceil\log_q(k+2)\rceil$ cells each, which we call ***index blocks***. Henceforth, it will be convenient to refer to the blocks of $k/2$ cells as ***parity blocks***, in order to distinguish them from the index blocks. Initially, the first k index blocks $\boldsymbol{u}_1, \boldsymbol{u}_2, \ldots, \boldsymbol{u}_k$ are set so that $\boldsymbol{u}_i = i$ (in the base-q number system), which reflects the fact that the information bits $v_0, v_1, \ldots, v_{k-1}$ are stored (in that order) in the first k live parity blocks. The next $m_1 - k$ index blocks are set to $(0, 0, \ldots, 0)$, thereby indicating that the corresponding (live) parity blocks are available to store information bits. The last $2(k-1) - m_1$ index blocks are set to $(q-1, q-1, \ldots, q-1)$ to indicate that the corresponding parity blocks are full (in fact, nonexistent). Finally, it is possible that in the process of enforcing $\texttt{parity}(\boldsymbol{x}_i) = v_{i-1}$ for the first k live parity blocks, some of these blocks become full (this happens iff $\mathtt{wt}(\boldsymbol{x}_i) = (k/2)(q-1) - 1$ and $v_i = 0$ at the end of the first stage, since $k/2$ is even by assumption). To account for this fact, we set the corresponding index blocks to $(q-1, q-1, \ldots, q-1)$. This completes the transition from the first stage to the second stage, which is invoked when the encoding map $\mathcal{E}_0$ produces the erasure symbol E.

Let us now summarize the foregoing discussion by giving a concise algorithmic description of the transition procedure.

Transition procedure $\mathcal{T}_1$: Partition the memory into $2\lfloor n/k \rfloor$ parity blocks of $k/2$ cells, and identify the $m_1 \leqslant 2(k-1)$ parity blocks $\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_{m_1}$ that are not full. If $m_1 < k$, output the erasure symbol E and terminate. Otherwise, set the $2(k-1)$ index blocks $\boldsymbol{u}_1, \boldsymbol{u}_2, \ldots, \boldsymbol{u}_{2k-2}$ as follows:

$$\boldsymbol{u}_i = \begin{cases} i & \text{for } i = 1,2,\ldots,k \\ 0 & \text{for } i = k+1, k+2, \ldots, m_1 \\ q^{\mu}-1 & \text{for } i = m_1{+}1, m_1{+}2, \ldots, 2k-2 \end{cases} \tag{2.3}$$

where $\mu = \lceil \log_q(k{+}2) \rceil$ is the number of cells in each index block, then record the information vector $(v_0, v_1, \ldots, v_{k-1})$ in the first k live parity blocks $\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_k$, as follows:

```
for ($i = 1$; $i \leqslant k$; $i = i+1$)
if (parity($\boldsymbol{x}_i$) $\neq v_{i-1}$)
{ increment($\boldsymbol{x}_i$); if (full($\boldsymbol{x}_i$)) $\boldsymbol{u}_i = q^{\mu}-1$;}
```

The function **full(*x*)** determines whether the given block $\boldsymbol{x}$ (which could be a parity block or an index block) is full. The function **increment(*x*)** increases by one the level of a cell (does not matter which) in the given live block.

During second-stage encoding and decoding, we will need to figure out for each active parity block $\boldsymbol{x}$ which of the k information bits it represents. To this end, we will have to find and read the index block $\boldsymbol{u}$ that *corresponds* to $\boldsymbol{x}$. How exactly is the correspondence between parity blocks and index blocks established? Note that, upon the completion of the transition procedure $\mathcal{T}_1$, there is the same number of live parity blocks and live index blocks; moreover, the j-th *live* index block corresponds to the j-th *live* parity block, for all j. The encoding procedure will make sure that this correspondence is preserved throughout the second stage: whenever a parity block becomes full, it will make the corresponding index block full as well.

We are now ready to present the encoding and decoding maps which are, again, specified in C-like pseudo-code notation.

Decoding map $\mathcal{D}_1$: The input is a cell-state vector $\boldsymbol{x} = (\boldsymbol{x}_1|\boldsymbol{x}_2|\cdots|\boldsymbol{x}_{2m}\|\,\boldsymbol{u}_1|\boldsymbol{u}_2|\cdots|\boldsymbol{u}_{2k-2})$, partitioned into $2m$ parity blocks, of $k/2$ cells each, and $2(k{-}1)$ index blocks. The output is the information vector $(v_0, v_1, \ldots, v_{k-1})$.

```
$(v_0, v_1, \ldots, v_{k-1})$ = $(0, 0, \ldots, 0)$;
for ($\ell = j = 1$; $j \leqslant 2m$; $j = j+1$)
{
  if (full($\boldsymbol{x}_j$)) continue;  // skip full blocks
  while (full($\boldsymbol{u}_\ell$)) $\ell = \ell+1$; // skip full blocks
  $i = \boldsymbol{u}_\ell$;  $\ell = \ell+1$;
  if ($i \neq 0$) $v_{i-1}$ = parity($\boldsymbol{x}_j$);
}
```

Given an index i of the bit that has changed, the encoding map $\mathcal{E}_1$ first tries to find an active parity block $\boldsymbol{x}$ that represents the i-th information bit. If such a block is found, it is incremented and checked for getting full (in which case the corresponding index block is set to $q^\mu - 1$). If not, another live parity block is allocated to represent the i-th information bit. If no more live parity blocks are available, the erasure symbol E is returned.

Encoding map $\mathcal{E}_1$: The input is a cell-state vector $\boldsymbol{x} = (\boldsymbol{x}_1|\boldsymbol{x}_2|\cdots|\boldsymbol{x}_{2m}\| \boldsymbol{u}_1|\boldsymbol{u}_2|\cdots|\boldsymbol{u}_{2k-2})$, partitioned into $2m$ parity blocks and $2(k-1)$ index blocks, and an index i of the information bit that changed. Its output is either a cell-state vector $\boldsymbol{y} = (\boldsymbol{y}_1|\boldsymbol{y}_2|\cdots|\boldsymbol{y}_{2m}\| \boldsymbol{u}'_1|\boldsymbol{u}'_2|\cdots|\boldsymbol{u}'_{2k-2})$ or the symbol E.

```
$(y_1|y_2|\cdots|y_{2m})$ = $(x_1|x_2|\cdots|x_{2m})$;
$(u'_1|u'_2|\cdots|u'_{2k-2})$ = $(u_1|u_2|\cdots|u_{2k-2})$;
for ($\ell$ = $j$ = 1; $j \leqslant 2m$; $j$ = $j+1$)
{
    if (full($x_j$)) continue;
    while (full($u_\ell$)) $\ell$ = $\ell+1$;
    if ($u_\ell$ == $i+1$)
    {
        increment($y_j$);
        if(full($y_j$)) $u'_\ell$ = $q^\mu - 1$;
        break;
    }
    else $\ell$ = $\ell+1$;
}
if ($j$ == $2m+1$)  // active block not found
for ($\ell$ = $j$ = 1; $j \leqslant 2m$; $j$ = $j+1$)
{
    if (full($x_j$)) continue;
    while (full($u_\ell$)) $\ell$ = $\ell+1$;
    if ($u_\ell$ == 0)
    {
        $u'_\ell$ = $i+1$;
        if (parity($x_j$) $\neq v_i$) increment($y_j$);
        if (full($y_j$)) $u'_\ell$ = $q^\mu - 1$;
        break;
    }
    else $\ell$ = $\ell+1$;
}
if ($j$ == $2m+1$)  // no more available live blocks
return E;
```

Note that when the second encoding stage terminates, there are at most $k-1$ parity blocks that are not full, comprising at most $k(k-1)/2$ cells (at most $k(k-1)(q-1)/2$ cell-levels).

Once the maps $\mathcal{D}_1$ and $\mathcal{E}_1$ are understood, it becomes clear that the same approach can be applied iteratively. The resulting flash code $\mathbb{C}^*$ will proceed, sequentially, through s different encoding stages $\mathcal{E}_0, \mathcal{E}_1, \ldots, \mathcal{E}_{s-1}$, where $s = \lceil \log_2 k \rceil$. In describing this code, we shall assume for the sake of simplicity that k is a power of two, that is $k = 2^s$. If not, the same code can be used to store $2^s > k$ information bits, of which the last $2^s - k$ are set to zero. Note that this will not change the order of the resulting write deficiency.

To accommodate the encoding maps $\mathcal{E}_1, \mathcal{E}_2, \ldots, \mathcal{E}_{s-1}$, we set aside for *each map* a batch of $2(k-1)$ index blocks, with each index block consisting of $\mu = \lceil \log_q(k+2) \rceil$ cells. The transition procedure $\mathcal{T}_r$ which bridges between the encoding maps $\mathcal{E}_{r-1}$ and $\mathcal{E}_r$ (for some $r \in \{2, 3, \ldots, s-1\}$) is identical to the transition procedure $\mathcal{T}_1$, except for the following differences:

D1. The r-th batch of index blocks is used; and

D2. The parity blocks consist of $k/2^r$ cells each.

In addition to **D1** and **D2**, the decoding/encoding maps $\mathcal{D}_r$ and $\mathcal{E}_r$ differ from $\mathcal{D}_1$ and $\mathcal{E}_1$ in that "$2m$" should be replaced by "$2^r m$" throughout, where m stands for $\lfloor n/k \rfloor$ as before. There are no other differences.

Theorem 2.5.1. *For $s = \lceil \log_2 k \rceil$, the write deficiency order of the flash code $\mathbb{C}^*$, which is defined by the sequence of decoding/encoding maps $\mathcal{D}_0, \mathcal{D}_1, \ldots, \mathcal{D}_{s-1}$ and $\mathcal{E}_o, \mathcal{E}_1, \ldots, \mathcal{E}_{s-1}$, is $O\big(qk \log^2 k / \log q\big)$.*

Proof. We consider the worst-case scenario for the number of cell levels that are either unused or "wasted" in the overall encoding procedure. As before, there are at most $k-1$ cells that are unused due to the partition into $\lfloor n/k \rfloor$ blocks, of exactly k cells each, at the very first encoding stage. These cells contribute at most $(q-1)(k-1)$ unused cell levels. The index blocks for the $s-1$ encoding maps $\mathcal{E}_1, \mathcal{E}_2, \ldots, \mathcal{E}_{s-1}$ contain $2(k-1)(s-1)\mu$ cells altogether, thereby wasting at most

$$2(q-1)(k-1)(s-1)\lceil \log_q(k+2) \rceil \;=\; O\left(\frac{qk \log^2 k}{\log q}\right) \tag{2.4}$$

cell levels. In each of the $s-1$ transition procedures, the situation $\texttt{parity(}x_i\texttt{)} \neq v_{i-1}$ can occur at most k times, and each time it occurs a single cell level is wasted. Finally, as in Theorem 2.4.1, when the encoding process $\mathcal{E}_o, \mathcal{E}_1, \ldots, \mathcal{E}_{s-1}$ terminates there are at most $k-1$ parity blocks that are not full and, in the worst case, each of them uses just one cell level. However, now these parity blocks contain only $\lceil k/2^{s-1} \rceil = 2$ cells each, and thus contribute at most $(k-1)(2q-3)$ unused cell levels. Putting all of this together, we find that at most

$$(q-1)(k-1)\Big(2(s-1)\lceil \log_q(k+2) \rceil + 3\Big) + k(s-1) \tag{2.5}$$

cell levels are wasted or left unused. Clearly, this expression is dominated by (2.4), and thus bounded by $O\big(qk\log^2 k/\log q\big)$. ■

For large q, the upper bound of $O\big(qk\log^2 k/\log q\big)$ on the deficiency of our scheme can be improved by using a more efficient "packaging" of index blocks in the flash memory. As before, we allocate a batch of $2(k-1)$ index blocks to each encoding stage except $\mathcal{E}_0$. But now, every index block will occupy $\mu' = \lceil \log_2(k+2) \rceil$ cells rather than $\mu = \lceil \log_q(k+2) \rceil$ cells, and the indices will be written in binary rather than in the base-q number system. This allows index blocks that correspond to successive encoding stages to be "stacked on top of each other" in the same memory cells. Specifically, the encoding stage $\mathcal{E}_1$ will use only cell levels 0 and 1 to record the indices in its index blocks. Once this stage is over, the index information recorded during $\mathcal{T}_1$ and $\mathcal{E}_1$ is no longer relevant, and the level of *all* the $2(k-1)\mu'$ cells in the $2(k-1)$ index blocks can be raised to 1. Thereafter, provided $q \geqslant 3$, the transition procedure $\mathcal{T}_2$ and the encoding map $\mathcal{E}_2$ can use cell levels 1 and 2 to record the relevant index information in the *same memory cells*. Proceeding in this manner, we can accommodate up to $q-1$ batches of index blocks in $2(k-1)\mu'$ memory cells. We shall refer to this indexing scheme as *stacked binary indexing* and denote the resulting flash code by $\mathbb{C}'$.

Theorem 2.5.2. *The write deficiency of the flash code $\mathbb{C}'$ defined by the sequence of decoding/encoding maps $\mathcal{D}_0, \mathcal{D}_1, \ldots, \mathcal{D}_{s-1}$ and $\mathcal{E}_0, \mathcal{E}_1, \ldots, \mathcal{E}_{s-1}$ that use stacked binary indexing is at most $O(qk\log k)$ if $q \geqslant \log_2 k$, and at most $O(k\log^2 k)$ otherwise.*

Proof. With stacked binary indexing, the number of cell levels wasted in all the $2(k-1)(s-1)$ index blocks is at most

$$2(q-1)(k-1)\left\lceil \frac{s-1}{q-1} \right\rceil \lceil \log_2(k+2) \rceil \tag{2.6}$$

Although for most values of k and q this is strictly less than (2.4), all the other terms in (2.5) are still dominated by (2.6). ■

Remark. If we need to store k *symbols*, rather than bits, over an alphabet of size $\ell > 2$, the same flash code can still be used, with an appropriate interface. With the linear womcode of [5], the ℓ-ary symbols can be represented using $\ell - 1$ bits in such a way that any symbol change corresponds to a single bit transition. The flash code $\mathbb{C}'$ can be now applied as is, and the resulting write deficiency is $O\big(\max\{q, \log_2 k\ell\}\, k\ell \log k\ell\big)$.

2.6 Flash Codes of Constant Rate

All of our results so far pertain to the case where $n \geqslant k^2$. In this section, we briefly examine the situation where both k and n are large, while $k/n = R$ for some constant $R < 1$. Observe that write deficiency $\delta(\mathbb{C}) = n(q{-}1) - t$ is *not* an appropriate figure of merit in this situation: a trivial code that guarantees $t = 0$ writes achieves write deficiency $n(q{-}1) = k(q{-}1)/R$, which is within a constant factor $2/R$ from the lower bound (2.1). Thus we will state our results in terms of the guaranteed number of writes t rather than the write deficiency $\delta(\mathbb{C})$.

If $q = 2$, we can easily guarantee $\Omega(n/\log k)$ writes as follows: partition the n cells into blocks of size $\lceil \log_2 k \rceil$ and each time an information bit changes, record its index in the next available block. For $q > 2$, the same method guarantees about $\lfloor n/\log_q k \rfloor = \Omega(n \log q/\log k)$ writes, but we can do better.

Let us partition the n cells into two groups: the *index group* consisting of $n - k$ cells and the *parity group* consisting of k cells. The index group is then subdivided into $m = \lfloor (n{-}k)/s \rfloor$ blocks, each consisting of $s = \lceil \log_2 k \rceil$ cells. The writing proceeds in $q - 1$ phases. During the first phase, every time an information bit changes, its index is recorded in binary (using cell levels 0 and 1) in the next available index block. After m writes, the first phase is over. We then copy the k information bits into the k cells of the parity group, and raise the level of all cells in the index group to 1. The second phase can now proceed using cell levels 1 and 2, and recording changes in information bits relative to the values stored in the parity group. At the end of the second phase, the current values of the k bits are recorded in the parity cells using levels 1 and 2. And so on. This simple coding scheme achieves

$$m(q-1) \;=\; \frac{n(q{-}1)(1-R)}{\log_2 k} \;=\; \Omega\left(\frac{nq}{\log k}\right) \tag{2.7}$$

writes (where the middle expression ignores ceilings/floors by assuming that k is a power of two and that $n - k$ is divisible by $\log_2 k$). If q is odd and $R \geqslant 0.415$, we can do a little better by using the ternary number system (cell levels $0, 1, 2$) in both the index group and the parity group. In this case, the size of the parity group is $\lceil k/\log_2 3 \rceil$ cells and $1 - R$ in (2.7) can be re-placed by $(\log_2 3 - R)/2$. Finally, for all $R \geqslant 0.755$ and $q - 1$ divisible by three, the quaternary alphabet is optimal, leading to a factor of $(2 - R)/3$ rather than $1 - R$ in (2.7).

2.7 Buffer Codes

Buffer codes were first presented by Bohossian et al. in [1]. In this family of codes, a buffer of r symbols has to be stored in n flash memory q-ary cells. After each write, the last r symbols that were written have to be recovered by the cell-state vector. The goal is to maximize the number of write symbols t the code guarantees without incurring a block erasure. In [1,9], a construction and upper bound of buffer codes that use one cell are given and a construction for arbitrary number of cells n, where $n \geqslant 2r$ is given as well.

2.7.1 Buffer Codes Definition

Let us first give a formal definition of buffer codes. We refer to the set of vectors in $\{0,\ldots,\ell-1\}^r$ as ***buffer vectors***. Similarly to flash codes, a buffer code $\mathbb{C}$ is also specified by an encoding map $\mathcal{E}$ and a decoding map $\mathcal{D}$. The ***decoding map*** $\mathcal{D}: \mathcal{A}_q^n \to \{0,\ldots,\ell-1\}^r$ assigns for each cell-state vector $\boldsymbol{x} \in \mathcal{A}_q^n$ its buffer vector $\mathcal{D}(\boldsymbol{x})$. Similarly, the ***encoding map*** $\mathcal{E}\colon \{0,\ldots,\ell-1\} \times \mathcal{A}_q^n \to \mathcal{A}_q^n \cup \{\mathsf{E}\}$ specifies to every symbol $a \in \{0,\ldots,\ell-1\}$ and cell-state vector $\boldsymbol{x} \in \mathcal{A}_q^n$, another cell-state vector $\boldsymbol{y} = \mathcal{E}(a,\boldsymbol{x})$ such that $y_j \geqslant x_j$ for all $1 \leqslant j \leqslant n$, $(\mathcal{D}(\boldsymbol{y}))_1 = a$ and for $2 \leqslant i \leqslant r$, $(\mathcal{D}(\boldsymbol{y}))_i = (\mathcal{D}(\boldsymbol{x}))_{i-1}$. In case such a $\boldsymbol{y} \in \mathcal{A}_q^n$ does not exist, then $\mathcal{E}(i,\boldsymbol{x}) = \mathsf{E}$. Thus, we give a formal definition of buffer codes.

Definition 2.7.1. *An $(n,\ell,r,t)_q$ buffer code $\mathbb{C}(\mathcal{D},\mathcal{E})$* ***guarantees*** *$t$* ***writes*** *if for all sequences of up to t symbol writes, the encoding map $\mathcal{E}$ does not produce the block erasure symbol E.*

2.7.2 Single-Cell Buffer Codes

In this section, we discuss the case where there is a single cell ($n = 1$) to store the buffer. A construction for this scenario where a binary buffer ($\ell = 2$) is stored was given in [1,9]. This construction guarantees at least $t = \left\lfloor \frac{q}{2^{r-1}} \right\rfloor + r - 2$ writes before a block erasure. An upper bound was given as well which asserts that for every buffer code with one cell the number of writes t has to satisfy

$$t \leqslant \left\lfloor \frac{q-1}{\ell^r - 1} \right\rfloor \cdot r + \lfloor ((q-1) \bmod (\ell^r - 1) + 1) \rfloor .$$

Let us show here another upper bound for such codes.

Theorem 2.7.2. *For any $(1,\ell,r,t)_q$ buffer code $\mathbb{C}$ such that $q \geqslant \ell^r$,*

$$t \leqslant \left\lfloor \frac{q - \ell^r}{\frac{1}{r}\sum_{d|r} \varphi(\frac{r}{d})\ell^d} \right\rfloor + r,$$

where φ is Euler's φ function.

Proof. Let $\mathbb{C}(\mathcal{D},\mathcal{E})$ be a $(1,\ell,r,t)_q$ buffer code. After $i \geqslant 1$ writes, for each $v \in \{0,1,\ldots,\ell-1\}^r$, let

$$S_i(v) = \{x \mid \text{there is a sequence of } j \leqslant i \text{ symbol writes ending in level } x \text{ and } \mathcal{D}(x) = v\},$$

$m_i(v) = \max_{x\in S_i(v)}\{x\}$ is the maximum cell level that is possible to reach after i symbol writes such that $\mathcal{D}(i) = v$, and

$$M_i = \sum_{v\in\{0,\ldots,\ell-1\}^r} |S_i(v)|.$$

Clearly, for all $i \leqslant t$, $M_i \leqslant q-1$. After r writes, it is possible to reach any of the ℓ^r different buffer vectors and thus $M_r \geqslant \ell^r - 1$.

Let $\mathcal{G}_{\ell,r}$ be the r-th order ℓ-ary de Bruijn graph [3]. Its vertices set is $\mathcal{V}_{\ell,r} = \{0,1,\ldots,\ell-1\}^r$ and its edges set is $\mathcal{E}_{\ell,r}$. Let $v_1, v_2 \in \{0,1,\ldots,\ell-1\}^r$ be two different buffer states. Note that if $(v_1,v_2) \in \mathcal{E}_{\ell,r}$ and $m_i(v_1) > m_i(v_2)$ then $m_{i+1}(v_2) > m_i(v_2)$ and therefore, the value of M_{i+1} increases by at least one level for every such an edge. In the de Bruijn graph every cycle has at least one edge $(v_1,v_2) \in \mathcal{E}_{\ell,r}$ such that $m_i(v_1) > m_i(v_2)$. Therefore, the number of new unused levels is at least as the number of disjoint vertex cycles in $\mathcal{G}_{\ell,r}$. This number is known to be $\frac{1}{r}\sum_{d|r}\varphi(\frac{r}{d})\ell^d$ [6, 15], and therefore

$$t \leqslant \left\lfloor \frac{q-\ell^r}{\frac{1}{r}\sum_{d|r}\varphi(\frac{r}{d})\ell^d} \right\rfloor + r.$$

■

Lemma 2.7.3. *The bound in Theorem 2.7.2 improves the bound in [1] for $q \geqslant \ell^r$. That is,*

$$\left\lfloor \frac{q-\ell^r}{\frac{1}{r}\sum_{d|r}\varphi(\frac{r}{d})\ell^d} \right\rfloor + r \leqslant \left\lfloor \frac{q-1}{\ell^r-1} \right\rfloor \cdot r + \left\lfloor \log_\ell\left(((q-1) \bmod (\ell^r-1))+1\right)\right\rfloor.$$

Proof. Note that

$$\frac{1}{r}\sum_{d|r}\varphi\left(\frac{r}{d}\right)\ell^d \geqslant \frac{\ell^r+\ell\varphi(r)}{r},$$

and therefore

$$\left\lfloor \frac{q-\ell^r}{\frac{1}{r}\sum_{d|r}\varphi(\frac{r}{d})\ell^d} \right\rfloor + r \leqslant \left\lfloor \frac{q-\ell^r}{\frac{\ell^r+\ell\varphi(r)}{r}} \right\rfloor + r = \left\lfloor \frac{q-\ell^r}{\ell^r+\ell\varphi(r)} \cdot r \right\rfloor + r = \left\lfloor \frac{q+\ell\varphi(r)}{\ell^r+\ell\varphi(r)} \cdot r \right\rfloor.$$

If we denote $q-1 = x(\ell^r-1)+y$, where $0 \leqslant y \leqslant \ell^r-1$, then

$$\left\lfloor \frac{q-\ell^r}{\frac{1}{r}\sum_{d|r}\varphi(\frac{r}{d})\ell^d} \right\rfloor + r \leqslant \left\lfloor \frac{q+\ell\varphi(r)}{\ell^r+\ell\varphi(r)}\cdot r \right\rfloor = \left\lfloor \frac{x(\ell^r-1)+y+1+\ell\varphi(r)}{\ell^r+\ell\varphi(r)}\cdot r \right\rfloor$$
$$= \left\lfloor \frac{x(\ell^r+\ell\varphi(r))-x+y+1-(x-1)\ell\varphi(r)}{\ell^r+\ell\varphi(r)}\cdot r \right\rfloor$$
$$= xr + \left\lfloor \frac{-x+y+1-(x-1)\ell\varphi(r)}{\ell^r+\ell\varphi(r)}\cdot r \right\rfloor \leqslant xr + \left\lfloor \frac{(y+1)r}{\ell^r} \right\rfloor.$$

Let us show that $\frac{(y+1)r}{\ell^r} \leqslant \log_\ell(y+1)$. That is, we show that $(y+1) \geqslant \ell^{\frac{(y+1)r}{\ell^r}}$ or

$$\left((y+1)^{\frac{1}{y+1}}\right)^{\ell^r} \geqslant \ell^r.$$

The function $f(x) = x^{\frac{1}{x}}$ is monotonically decreasing for $x \geqslant 1$ and since $y \leqslant \ell^r - 1$, we get

$$\left((y+1)^{\frac{1}{y+1}}\right)^{\ell^r} \geqslant \left((\ell^r)^{\frac{1}{\ell^r}}\right)^{\ell^r} = \ell^r.$$

All together we get that

$$\left\lfloor \frac{q-\ell^r}{\frac{1}{r}\sum_{d|r}\varphi(\frac{r}{d})\ell^d} \right\rfloor + r \leqslant xr + \left\lfloor \frac{(y+1)r}{\ell^r} \right\rfloor$$
$$\leqslant xr + \lfloor \log_\ell(y+1) \rfloor = \left\lfloor \frac{q-1}{\ell^r-1} \right\rfloor \cdot r + \left\lfloor \log_\ell\left(((q-1) \bmod (\ell^r-1))+1\right) \right\rfloor.$$

■

2.7.3 Multiple-Cells Buffer Codes

In [1,9], a buffer code construction is given for $\ell = 2$ and arbitrary n, q, r, where $n \geqslant 2r$. This construction guarantees $t = (q-1)(n-2r+1)+r-1$ writes. In this Section, we show how to improve this construction such that its number of writes is $t = (q-1)(n-r)$.

In case that $q = 2$, the construction in [1, 9] guarantees $n-r$ writes. The encoding procedure is performed in such a way that after i writes, $1 \leqslant i \leqslant n-r$, the buffer is located between the $(i+1)$-st and $(i+r)$-th cells, where the first bit of the buffer memory is stored in the $(i+r)$-th cell and the last bit is stored in the $(i+1)$-st cell. If $q > 2$, then the construction uses the cell levels by a "layer by layer" approach. That is, first the layer of levels 0 and 1 is used, then the layer of levels 1 and 2 is used, and so on. In the transition from a the layer of levels $i-1, i$ to the layer of levels $i, i+1$, first all the cells are reset to level i and the buffer is written in the new layer of levels $i, i+1$. Then, it is possible to continue writing in this layer.

Basically, on each layer, it is possible to write $n-r$ times. However, when a new layer is used, then first the buffer from the previous layer is copied and then is written in the new layer. Hence, it is possible to have only $(n-2r+1)$ more writes in the new layer and the total number of writes is

$$n-r+(q-2)(n-2r+1)=(q-1)(n-2r+1)+r-1.$$

The transition between these consecutive layers is not performed efficiently and our improvement here show how it is possible to write $n-r$ times on each layer such that the total number of writes is $t=(q-1)(n-r)$. We present this construction by its encoding and decoding maps specification.

Decoding map $\mathcal{D}_{\text{buf}}$: The input to this map is a cell-state vector $\boldsymbol{x}=(x_1,x_2,\ldots,x_n)$. The output is the corresponding information buffer vector $(v_1,v_2,\ldots,v_r)$.

```
$m$ = max($x_1, x_2, \ldots, x_n$);
$n_m$ = find_repeat($m, x_1, x_2, \ldots, x_n$);
if($n_m \geqslant r$)
  for($i$ = 1; $i \leqslant r$; $i$ = $i$ + 1)
    $v_i$ = $x_{r+n_m-i+1}$ - $m$;
else {
  for($i$ = 1; $i \leqslant n_m$; $i$ = $i$ + 1)
    $v_i$ = $x_{r+n_m-i+1}$ - $m$;
  for($i$ = $n_m$ + 1; $i \leqslant r$; $i$ = $i$ + 1)
    $v_i$ = $x_{n+n_m-i+1}$ - ($m$ - 1);}
```

The function **max**$(x_1,x_2,\ldots,x_n)$ simply returns the maximum value of the cells $x_1,x_2,\ldots,x_n$. The function **find_repeat**$(m,x_1,x_2,\ldots,x_n)$ returns the number of times the value m repeats in the cells $x_1,x_2,\ldots,x_n$. If the value of n_m is at least r then the buffer is stored between the (n_m+1)-st and (n_m+r)-th cells, and the buffer values are calculated by subtracting m from the value of each cell. If the value of n_m is less than r then the buffer is stored cyclically in two cell groups: the last $r-n_m$ cells and the n_m cells in locations $r+1,\ldots,r+n_m$. In the first group, the buffer values are given by subtracting $m-1$ from the cells' value and in the second group by subtracting m from the cells' value.

Encoding map $\mathcal{E}_{\text{buf}}$: The input to this map is a cell-state vector $\boldsymbol{x}=(x_1,x_2,\ldots,x_n)$, and a new bit b. Its output is either a cell-state vector $\boldsymbol{y}=(y_1,y_2,\ldots,y_n)$ or the erasure symbol E.

```
$(y_1, y_2, \ldots, y_n)$ = $(x_1, x_2, \ldots, x_n)$;
$m$ = max($x_1, x_2, \ldots, x_n$);
$n_m$ = find_repeat($m, x_1, x_2, \ldots, x_n$);
if($m$ == 0) { // if this is the first write
  if($b$ == 1) $y_{r+1}$ = 1;
  else $y_1$ = 1; }
if($n_m$ == $n - r$) { // first write in this layer
  for($i$ = 1; $i \leqslant n - r + 1$; $i$ = $i$ + 1)
    $y_i$ = $m$;
  if($b$ == 1) $y_{r+1}$ = $m$ + 1;
  else $y_1$ = $m$ + 1; }
if($n_m < n - r$) { // not the first write in this layer
  $y_{r+n_m+1}$ = $y_{r+n_m+1}$ + $b$;
  if($b$ == 0)
    for($i$ = 1; $i \leqslant n_m + r$; $i$ = $i$ + 1)
      if($y_i$ == $m$ - 1) {
        $y_{r+n_m+1}$ = $y_{r+n_m+1}$ + 1; break; } }
if($n_m \leqslant r - 1$) // one of first $r - 1$ writes in this layer
    $y_{n-r+1+n_m}$ = $m$ - 1;
```

On the first write, according to the bit value b the first or the $(r+1)$-st cell changes its value to one. On the first write on each layer, the first $n-r+1$ cells are increased to level m, and then the first or the $(r+1)$-st cell is increased by one level, according to the bit value b. For all other writes, if the value if b is one then we simply increase the $(r+n_m+1)$-st cell by one level, and otherwise we increase the first cell of level $m-1$ by one level. Finally, if it is one of the first $r-1$ writes in each level, then we need to update the last cell that stores the buffer to level $m-1$ since it no longer stores the buffer and thus its level has to be updated. The next example demonstrates how this construction works.

Example 2.7.1. In this example, we show how the last construction works for $n = 11, q = 3, \ell = 2$ and $r = 4$, so the number of writes is $2 \cdot (11-4) = 14$. The sequence of write bits is $1,1,0,0,1,0,0,1,1,1,0,1,1,0$ and they are performed as follows. The underlined cells represent the cells that store the buffer on each write.

Written Bit	Buffer State	Cell State Vector
	$(0,0,0,0)$	$(\underline{0,0,0,0},0,0,0,0,0,0,0)$
1	$(0,0,0,1)$	$(0,\underline{0,0,0,1},0,0,0,0,0,0)$
1	$(0,0,1,1)$	$(0,0,\underline{0,0,1,1},0,0,0,0,0)$
0	$(0,1,1,0)$	$(1,0,0,\underline{0,1,1,0},0,0,0,0)$
0	$(1,1,0,0)$	$(1,1,0,0,\underline{1,1,0,0},0,0,0)$
1	$(1,0,0,1)$	$(1,1,0,0,1,\underline{1,0,0,1},0,0)$
0	$(0,0,1,0)$	$(1,1,1,0,1,1,\underline{0,0,1,0},0)$
0	$(0,1,0,0)$	$(1,1,1,1,1,1,0,\underline{0,1,0,0})$
1	$(1,0,0,1)$	$(1,1,1,1,\underline{2},1,1,1,\underline{1,0,0})$
1	$(0,0,1,1)$	$(1,1,1,1,\underline{2,2},1,1,1,\underline{0,0})$
1	$(0,1,1,1)$	$(1,1,1,1,\underline{2,2,2},1,1,1,\underline{0})$
0	$(1,1,1,0)$	$(2,1,1,1,\underline{2,2,2,1},1,1,1)$
1	$(1,1,0,1)$	$(2,1,1,1,2,\underline{2,2,1,2},1,1)$
1	$(1,0,1,1)$	$(2,1,1,1,2,2,\underline{2,1,2,2},1)$
0	$(0,1,1,0)$	$(2,1,1,1,2,2,2,\underline{1,2,2,1})$

Next, we prove the correctness of the construction.

Lemma 2.7.4. *After $s = x(n-r)+y$, where $1 \leqslant y \leqslant n-r$, the maximum cell level is $x+1$ and there are y cells in level $x+1$.*

Proof. According to the encoding map $\mathcal{E}_{\text{buf}}$, the maximum cell level increases every $n-r$ write, on the $(i(n-r)+1)$-st write, for $0 \leqslant i \leqslant q-2$. Therefore, after s writes, the maximum cell value is $x = \lceil \frac{s}{n-r} \rceil$. If $y = 1$ then the maximum cell value is $x+1$ and we can see that exactly one cell changes its value to $x+1$. For all other writes, the maximum cell value does not change and exactly one cell changes its value to the maximum cell value which is $x+1$. ■

Theorem 2.7.5. *The buffer code $\mathbb{C}(\mathcal{D}_{buf}, \mathcal{E}_{buf})$ stores the buffer successfully and guarantees $t = (q-1)(n-r)$ writes.*

Proof. According to Lemma 2.7.4, after $t = (q-1)(n-r)$ writes the maximum cell level does not reach level q and hence there is no need to erase the block of cells. We prove the correctness of the encoding and decoding maps to store the correct value of the buffer by induction on the number of writes s. This is done by proving that for all $1 \leqslant s \leqslant t$, such that $s = x(n-r) + y$, where $1 \leqslant y \leqslant n-r$, the buffer $(v_1, \ldots, v_r)$ is calculated successfully according to the decoding rules of the decoding map:

1. If $y \geqslant r$ then for $1 \leqslant i \leqslant r$, $v_i = x_{r+y-i+1} - m$.

2. If $y < r$ then for $1 \leqslant i \leqslant y$, $v_i = x_{r+y-i+1} - m$ and for $y+1 \leqslant i \leqslant r$, $v_i = x_{r+y-i+1} - (m-1)$.

It is straightforward to verify that after the first write the memory successfully stores the buffer. Assume the assertion is correct after the s-th write, where $1 \leqslant s = x(n-r)+y \leqslant t-1$, $1 \leqslant y \leqslant n-r$. Assume that the new bit to be written to the buffer on the $(s+1)$-st write is b and let us consider the following cases:

1. If $y = n-r$, then on the $(s+1)$-st write in the encoding map the value of n_m is $n-r$. Thus the first $n-r+1$ cells change their value to $m = x$, the value of the last $r-1$ cells do not change, and if $b = 1$ then $y_{r+1} = m+1$, and otherwise $y_1 = m+1$. Therefore, the new value of the buffer is also given according to the decoding rules.

2. If $y < n-r$, then $n_m = y < n-r$, and the value of the $(r+n_m+1)$-st cell increases by b so the buffer is shifted one place to the right and it stores its updated value. If $b = 0$, then we increase one the first n_m+1 cells by one level. Note that $n_m = y$ and there are exactly y cells with the maximum value so we can always find a cell of value less than m and increase it to be m. Then, again the buffer is stored according to the above decoding rules.

■

2.8 Conclusion and Open Problems

Rewriting codes for flash memories are of high importance as they allow to increase the lifetime of the memory. Examples of such codes are flash codes [8] and buffer codes [1]. Our main contribution in this chapter is an efficient construction of flash codes that support the storage of any number of bits. We show that the write deficiency order of the code is $O(k \log k \cdot \max\{\log k, q\})$, which is an improvement upon equivalent constructions in [9,9,17]. The upper bound in [8] on the guaranteed number of writes implies that the order of the lower bound on the deficiency is $O(kq)$. Therefore, there is a gap, which we believe can be reduced, between the write deficiency orders of our construction and the lower bound. For buffer codes, we showed how to improve an upper bound on the number of writes in case one cell is used to store the buffer. If there are multiple cells, we showed an improved construction upon the one presented in [1,9].

Acknowledgment

This chapter is in part a reprint of the material in the papers: E. Yaakobi, P.H. Siegel, and J.K. Wolf, “Buffer Codes for Multi-Level Flash Memory,” *IEEE International Symposium on Information Theory*, Poster session, Toronto, Canada, July 2008, and H. Mahdavifar, P.H. Siegel, A. Vardy, J.K. Wolf, and E.Yaakobi, “A Nearly Optimal Construction of Flash Codes,” *Proc. IEEE International Symposium on Information Theory*, pp. 1239–1243, Seoul, Korea, June 2009.

Bibliography

[1] V. Bohossian, A. Jiang, and J. Bruck, “Buffer coding for asymmetric multilevel memory,” *Proc. IEEE International Symposium on Inform. Theory*, pp. 1186–1190, Nice, France, June 2007.

[2] G.D. Cohen, P. Godlewski, and F. Merkx, “Linear binary code for write-once memories,” *IEEE Trans. Inform. Theory*, vol. 32, no. 5, pp. 697–700, September 1986.

[3] N.G. de Bruijn, “A combinatorial problem”, *Proc. K. Ned. Akad. Wet. Ser. A*, Vol. 49, pp. 758–764, 1946.

[4] A. Fiat and A. Shamir, “Generalized write-once memories,” *IEEE Trans. Inform. Theory*, vol. 30, pp. 470–480, September 1984.

[5] H. Finucane, Z. Liu, and M. Mitzenmacher, “Designing floating codes for expected performance,” *Proc. 46-th Allerton Conf. Communication, Control and Computing*, pp. 1389–1396, Monticello, IL, September 2008.

[6] S.W. Golomb, *Shift Register Sequences*, revised edition, Aegean Park Press, Laguna Hills, CA, 1982.

[7] A. Jiang, “On the generalization of error-correcting WOM codes,” *Proc. IEEE International Symposium on Information Theory*, pp. 1391–1395, Nice, France, June 2007.

[8] A. Jiang, V. Bohossian, and J. Bruck, “Floating codes for joint information storage in write asymmetric memories,” *Proc. IEEE International Symposium on Information Theory*, pp. 1166–1170, Nice, France, June 2007.

[9] A. Jiang, V. Bohossian, and J. Bruck, “Rewriting codes for joint information storgae in flash memories,” *IEEE Trans. Inform. Theory*, vol. 56, no. 10, pp. 5300–5313, October 2010.

[10] A. Jiang and J. Bruck, “Joint coding for flash memory storage,” *Proc. IEEE International Symposium on Information Theory*, pp. 1741–1745, Toronto, Canada, July 2008.

[11] A. Jiang, M. Landberg, M. Schwartz, and J. Bruck, “Universal rewriting in constrained memories,” *Proc. IEEE International Symposium on Information Theory*, pp. 1219–1223, Seoul, Korea, July 2009.

[12] A. Jiang, R. Mateescu, M. Schwartz, and J. Bruck, “Rank modulation for flash memories,” *IEEE Trans. Inform. Theory*, vol. 55, no. 6, pp. 2659–2673, October 2010.

[13] A. Jiang, M. Schwartz, and J. Bruck, “Correcting Charge-constrained Errors in The Rank Modulation Scheme,” *IEEE Trans. Inform. Theory*, vol. 56, no. 5, pp. 2112–2120, May 2010.

[14] H. Mahdavifar, P.H. Siegel, A. Vardy, J.K. Wolf, and E. Yaakobi, “A nearly optimal construction of flash codes,” *Proc. IEEE Int. Symp. Inform. Theory*, pp. 1239–1243, Seoul, Korea, July 2009.

[15] J. Mykkeltveit, “A proof of Golomb’s conjecture for the de Bruijn graph”, *Journal of Combinatorial Theory (B)*, vol. 13, pp. 40–45, 1972.

[16] R.L. Rivest and A. Shamir, “How to reuse a write-once memory,” *Inform. and Control*, vol. 55, nos. 1–3, pp. 1–19, December 1982.

[17] E. Yaakobi, A. Vardy, P.H. Siegel, and J.K. Wolf, “Multidimensional flash codes,” *Proc. 46-th Allerton Conference on Communication, Control and Computing*, pp. 392–399, Monticello, IL, September 2008.

Chapter 3

Codes for Write-Once Memories

3.1 Introduction

Write-once-memory (WOM) codes were introduced in 1982 by Rivest and Shamir [5]. They make it possible to record binary data more than once in a so-called write-once storage medium, such as a punch card or ablative optical disk. These media can be represented as a collection of write-once bit locations, each of which initially represents a bit value 0 that can be ***irreversibly*** overwritten with a bit value 1. A WOM-code allows the reuse of a write-once medium by introducing redundancy into the recorded bit sequence and, in subsequent write operations, observing the state of the medium before determining how to update the contents of the memory with a new bit sequence.

A simple example, presented in [5], enables the recording of two bits of information in 3 memory elements, twice. The encoding and decoding rules for this WOM-code are described in a tabular form in Table 3.1. It is easy to verify that after the first 2-bit data vector is encoded into a 3-bit codeword, if the second 2-bit data vector is different from the first, the 3-bit codeword into which it is encoded does not change any code bit 1 into a code bit 0, ensuring that it can be recorded in the write-once medium.

Table 3.1: A WOM-code Example

Data bits	First write	Second write (if data changes)
00	000	111
10	100	011
01	010	101
11	001	110

Flash memories impose constraints on recording that are similar to those associated with write-once memories. As explained in Chapter 1, while it is fast and simple to increase a flash memory cell level, reducing its level requires a long and cumbersome operation of first erasing its entire containing block ($\sim 10^6$ cells) and only then programming the cells. A WOM-code can be applied in this context to enable additional writes without first having to erase the entire block. The deferral of a block erasure is beneficial to the lifetime of the device. The cost associated with this increase in the endurance is the redundancy and the additional complexity associated with the encoding and decoding processes. For more details on the implementation of WOM-codes in flash memories, the reader is referred to [3, 22].

The most fundamental problem in the WOM model is to maximize the total amount of information that can be written into n memory cells in t writes, while preserving the constraint that on each write one can only change cells in the zero state to the one state. The first WOM-code construction, presented by Rivest and Shamir, was designed for the storage of two bits twice using only three cells [5]. In their work, Rivest and Shamir also reported on more WOM-code constructions, including tabular WOM-codes and "linear" WOM-codes. Merkx constructed WOM-codes based on projective geometry [13]. In [2], using binary linear codes, Cohen et al. introduced a "coset-coding" technique that is used to construct WOM-codes, and in [5], an improvement to one of the constructions in [2] was given by Godlewski. Recently, position modulation codes were introduced by Wu and Jiang in order to construct multiple-write WOM-codes [18]. Wu found WOM-codes for two writes in [17] which improved the best rate previously known.

Wolf et al. discussed the WOM-codes problem from its information-theoretic point of view [16]. In [3], the WOM model has been generalized for multi-level cells and information theory limits and code constructions for constrained sources were studied. Heegard studied the capacity of a WOM and a noisy WOM in [6], and Fu and Han Vinck found the capacity of a non-binary WOM [4]. Error-correcting WOM-codes were first studied in [20], [21] and more constructions were recently given in [29]. Jiang discussed in [7] the generalization of error-correcting WOM-codes for the flash/floating codes model [8, 9, 12].

3.2 Preliminaries

In this work, the memory elements, called ***cells***, have two states: zero and one. At the beginning, all the cells are in their zero state. A cell can change its state from zero to one. This operation is irreversible in the sense that a cell cannot change its state from one to zero unless the entire memory is erased. The ***memory-state vectors*** are all the binary vectors of length n, $\{0,1\}^n$. For two memory-state vectors $\boldsymbol{c}, \boldsymbol{c}' \in \{0,1\}^n$, we denote by $\boldsymbol{c} \geqslant \boldsymbol{c}'$, if and only if $\boldsymbol{c}_i \geqslant \boldsymbol{c}'_i$ for all $1 \leqslant i \leqslant n$ and we say that $\boldsymbol{c}$ covers $\boldsymbol{c}'$.

Definition 3.2.1. *An $[n, t; M_1, \dots, M_t]$* ***t-write WOM-code*** *$\mathcal{C}$ is a coding scheme which consists of n cells and t pairs of encoding and decoding maps, denoted by $\mathcal{E}_i$ and $\mathcal{D}_i$ for $1 \leqslant i \leqslant t$. The t-write WOM-code $\mathcal{C}$ satisfies the following properties:*

1. *$\mathcal{E}_1 : \{1, \dots, M_1\} \to \{0,1\}^n$,*

2. *For $2 \leqslant i \leqslant t$,*
$$\mathcal{E}_i : \{1, \dots, M_i\} \times \{0,1\}^n \to \{0,1\}^n,$$
such that, for all $(m, \boldsymbol{c}) \in \{1, \dots, M_i\} \times \{0,1\}^n$,
$$\mathcal{E}_i(m, \boldsymbol{c}) \geqslant \boldsymbol{c}.$$

3. *For $1 \leqslant i \leqslant t$,*
$$\mathcal{D}_i : \{0,1\}^n \to \{1, \dots, M_i\},$$
such that $\mathcal{D}_1(\mathcal{E}_1(m)) = m$ for all $m \in \{1, \dots, M_1\}$, and for $2 \leqslant i \leqslant t$, $\mathcal{D}_i(\mathcal{E}_i(m, \boldsymbol{c})) = m$ for all $(m, \boldsymbol{c}) \in \{1, \dots, M_i\} \times \{0,1\}^n$.

The ***sum-rate*** *of a t-write WOM-code $\mathcal{C}$ is defined to be*
$$\mathcal{R}_{sum}(\mathcal{C}) = \frac{\sum_{i=1}^{t} \log_2 M_i}{n}.$$

Remark 3.2.1. We assume that the write number on each write is known. This knowledge does not affect the sum-rate. Indeed, assume that there exists an $[n, t; M_1, \dots, M_t]$ t-write WOM-code $\mathcal{C}$ where the write number is known. Assume also that the sum-rate of $\mathcal{C}$ is $\mathcal{R}_{\text{sum}}(\mathcal{C}) = \frac{\sum_{i=1}^{t} \log_2 M_i}{n}$. It is possible to change this WOM-code to an $[Nn + t, t; M_1^N, \dots, M_t^N]$ t-write WOM-code $\mathcal{C}'$ by having N blocks of the t-write WOM-code $\mathcal{C}$ and t more cells indicating the write number. Then, the sum-rate of $\mathcal{C}'$ is
$$\begin{aligned} \mathcal{R}_{\text{sum}}(\mathcal{C}') &= \frac{\sum_{i=1}^{t} \log_2 M_i^N}{Nn + t} = \frac{N \sum_{i=1}^{t} \log_2 M_i}{Nn + t} \\ &= \frac{N \sum_{i=1}^{t} \log_2 M_i}{Nn} \cdot \frac{Nn}{Nn + t} = \frac{\mathcal{R}_{\text{sum}}(\mathcal{C})}{1 + \frac{t}{Nn}}. \end{aligned}$$
Therefore, for N large enough it is possible to achieve the sum-rate of the t-write WOM-code $\mathcal{C}$. For simplicity, we will assume in this work that the write number is known in the encoding process.

While there are different ways to analyze the efficiency of WOM-codes, we find that the appropriate figure of merit is to analyze the sum-rate under the assumption of a fixed number of writes. In general, the more writes the WOM-code can support, the better the sum-rate it can achieve. The goal is to give upper and lower bounds on the sum-rates of WOM-codes while fixing the number of writes t.

Note that there are two different problems we can address when analyzing WOM-codes: we either require that on all writes the written message comes from an alphabet of fixed size M, or we allow the size of the alphabet of possible messages to vary from one write to another. In the first case we say that the WOM-code is ***fixed-rate***, whereas in the second case we say it is ***unrestricted-rate***.

3.3 Previous Work

It is proved in [4] and [6] that the capacity region of a binary t-write WOM-code is

$$\begin{aligned} C_t = \Big\{ (\mathcal{R}_1, \ldots, \mathcal{R}_t) \mid \mathcal{R}_1 \leqslant h(p_1), \mathcal{R}_2 \leqslant (1-p_1)h(p_2), \\ \ldots, \mathcal{R}_{t-1} \leqslant \Big(\textstyle\prod_{i=1}^{t-2}(1-p_i) \Big) h(p_{t-1}), \mathcal{R}_t \leqslant \textstyle\prod_{i=1}^{t-1}(1-p_i), \\ \text{where } 0 \leqslant p_1, \ldots, p_{t-1} \leqslant 1/2 \Big\}. \end{aligned} \tag{3.1}$$

It has been shown that all points in the capacity region can be achieved by random coding with unrestricted-rate WOM- codes. The sum-rate of the WOM-code is given by

$$\mathcal{R} = \sum_{j=1}^{t} \mathcal{R}_j = h(p_1) + \sum_{j=2}^{t-1} \Big(\prod_{i=1}^{j-1} (1-p_i) h(p_j) \Big) + \prod_{i=1}^{t-1} (1-p_i).$$

It is proved in [6] that the sum-rate is maximized when

$$p_j = \frac{1}{2+t-j},$$

for $1 \leqslant j \leqslant t-1$, and the maximum sum-rate is $\log_2(t+1)$. For example, for $t = 2$, the maximum sum-rate, $\log_2 3$, is achieved for $p_1 = 1/3$. Intuitively, this upper bound is plausible. During the course of the t writes, a particular cell can be programmed at some time $j \in 1, \ldots, t$ or not programmed at all. Thus there are $t+1$ possible scenarios, so the amount of information that can be stored in each cell is no greater that $\log_2(t+1)$. Of course, the result above indicates that this is a tight upper bound.

The case of fixed-rate WOM-codes was discussed in [6]. In this setting, we consider those points on the boundary of the capacity region C_t satisfying $\mathcal{R}_1 = \cdots = \mathcal{R}_t$. The maximum sum-rate, denoted by $\mathcal{R}_U^F(t)$, is given by the recursion in the following Theorem [6].

Table 3.2: Upper bounds on the sum-rate of fixed-rate WOM-codes

t	$\mathcal{R}_U^F(t)$	t	$\mathcal{R}_U^F(t)$
1	1	6	2.712
2	1.546	7	2.9001
3	1.9368	8	3.0664
4	2.2436	9	3.2157
5	2.4965	10	3.352

Theorem 3.3.1. *The values of $\mathcal{R}_U^F(t)$ for $t \geqslant 1$ satisfy the following recursive formula:*

$$\begin{aligned}\mathcal{R}_U^F(1) &= 1\\ \mathcal{R}_U^F(t+1) &= (t+1)\cdot \mathit{root}\left\{h\left(\frac{z}{\mathcal{R}_U^F(t)/t}\right)-z\right\},\end{aligned}$$

where $\mathit{root}\{f(z)\}$ is the minimum positive value of z such that $f(z)=0$.

As in the case of unrestricted-rate WOM-codes, the upper bound $\mathcal{R}_U^F(t)$ is tight. Using the recursion in the theorem, we obtain the following results for $\mathcal{R}_U^F(t)$ in Table 3.2.

The upper bounds on the sum-rates for fixed- and unrestricted-rate presented above have been shown to be achievable in theory. However, finding specific WOM-code constructions that achieve these maximum possible sum-rates remains an open problem. In the rest of this section, we give a brief summary of the highest known sum-rates that were achieved by previously published WOM-code constructions.

Rivest and Shamir, 1982 [5]

Rivest and Shamir constructed the first $[3,2;4,4]$ WOM-code ($\mathcal{R}_{\text{sum}} = 1.3333$), that stores two bits twice using only three cells. They constructed other WOM-codes, including a $[7,2;26,26]$ WOM-code ($\mathcal{R}_{\text{sum}} = 1.343$) which has a slightly better sum-rate, a $[7,3;8,8,8]$ WOM-code ($\mathcal{R}_{\text{sum}} = 1.2857$), and a $[7,5;4,4,4,4,4]$ WOM-code ($\mathcal{R}_{\text{sum}} = 1.4286$). They also described construction methods for various classes of WOM-codes, including tabular WOM-codes and "linear" WOM-codes. In their paper, they also mentioned specific WOM-codes as well as some classes of WOM-codes designed by others, with the following parameters:

1. $[5,3;5,5,5]$ WOM-code ($\mathcal{R}_{\text{sum}} = 1.3932$), by David Klaner.
2. $[7,4;7,7,7,7]$ WOM-code ($\mathcal{R}_{\text{sum}} = 1.6042$), by David Leavitt.
3. $[12,3;65,81,64]$ WOM-code ($\mathcal{R}_{\text{sum}} = 1.5302$), by James B. Saxe.
4. $[n, n/2-1; n/2, n/2-1, n/2-2, \ldots, 2]$ WOM-code, n even ($\mathcal{R}_{\text{sum}} \approx \frac{\log_2 n}{2}$ for n large enough), by James B. Saxe.

Merkx, 1984 [13]

Merkx constructed WOM-codes based on projective geometry codes. Parameters of some of his WOM-codes are:

1. $[7,4;7,7,7,7]$ WOM-code ($\mathcal{R}_{\text{sum}} = 1.6042$).
2. $[31,10;31,31,31,31,31,31,31,31,31,31]$ WOM-code ($\mathcal{R}_{\text{sum}} = 1.5981$).
3. $[7,4;8,7,8,8]$ WOM-code ($\mathcal{R}_{\text{sum}} = 1.6868$).
4. $[7,4;8,7,11,8]$ WOM-code ($\mathcal{R}_{\text{sum}} = 1.7524$).
5. $[8,4;8,14,11,8]$ WOM-code ($\mathcal{R}_{\text{sum}} = 1.6583$).
6. $[16,7;16,16,16,16,16,16,16]$ WOM-code ($\mathcal{R}_{\text{sum}} = 1.75$).
7. $[15,7;15,15,15,15,15,15,15]$ WOM-code ($\mathcal{R}_{\text{sum}} = 1.8232$).

Cohen, Godlewski, and Merkx, 1986 [2]

Cohen et al. introduced the “coset-coding” technique, which uses binary linear codes in the construction of WOM-codes. This approach yielded WOM-codes with the following parameters:

1. $[23,3;2^{11},2^{11},2^{11}]$ WOM-code ($\mathcal{R}_{\text{sum}} = 1.4348$).
2. $[3,2;4,4]$ WOM-code ($\mathcal{R}_{\text{sum}} = 1.3333$).
3. $[7,3;8,8,8]$ WOM-code ($\mathcal{R}_{\text{sum}} = 1.2857$).
4. $[2^r-1,2^{r-2}+2;2^r,2^r,\ldots,2^r]$ WOM-code ($\mathcal{R}_{\text{sum}} = \frac{r(2^{r-2}+2)}{2^r-1}$), for $r \geqslant 4$.

Godlewski, 1987 [5]

Godlewski improved upon the last result in [2] by constructing WOM-codes with parameters:

1. $[2^r-1,2^{r-2}+2^{r-4}+2;2^r,2^r,\ldots,2^r]$ WOM-code ($\mathcal{R}_{\text{sum}} = \frac{r(2^{r-2}+2^{r-4}+2)}{2^r-1}$), for $r \geqslant 4$.

Wu and Jiang, 2009 [18]

Recently, position modulation codes were used by Wu and Jiang in order to construct multiple-write WOM-codes. Their construction can produce many WOM-codes, among them WOM-codes with the following parameters:

Table 3.3: Prior best-known sum-rates for unrestricted-rate and fixed-rate WOM-codes.

Number of Writes	Best Prior (unrestricted)	Upper Bound (unrestricted)	Best Prior (fixed)	Upper Bound (fixed)
2	1.3707 [17]	1.585	1.343 [5]	1.546
3	1.5302 [5]	2	1.4348 [2]	1.9368
4	1.7524 [13]	2.3219	1.6042 [5, 13]	2.2436
5	1.7524 [13]	2.585	1.6279 [18]	2.4965
6	1.7524 [13]	2.8074	1.7143 [18]	2.712
7	1.8232 [13]	3	1.8232 [13]	2.9001
8	1.8824 [18]	3.1699	1.8824 [18]	3.0664
9	1.9535 [18]	3.3219	1.9535 [18]	3.2157
10	2.0144 [18]	3.4594	2.0144 [18]	3.352

1. $[172, 5; 2^{56}, 2^{56}, 2^{56}, 2^{56}, 2^{56}]$ WOM-code ($\mathcal{R}_{\text{sum}} = 1.6279$).
2. $[196, 6; 2^{56}, 2^{56}, 2^{56}, 2^{56}, 2^{56}, 2^{56}]$ WOM-code ($\mathcal{R}_{\text{sum}} = 1.7143$).
3. $[216, 7; 2^{56}, 2^{56}, 2^{56}, 2^{56}, 2^{56}, 2^{56}, 2^{56}]$ WOM-code ($\mathcal{R}_{\text{sum}} = 1.8148$).
4. $[238, 8; 2^{56}, 2^{56}, 2^{56}, 2^{56}, 2^{56}, 2^{56}, 2^{56}, 2^{56}]$ WOM-code ($\mathcal{R}_{\text{sum}} = 1.8824$).
5. $[258, 9; 2^{56}, 2^{56}, 2^{56}, 2^{56}, 2^{56}, 2^{56}, 2^{56}, 2^{56}, 2^{56}]$ WOM-code ($\mathcal{R}_{\text{sum}} = 1.9535$).
6. $[278, 10; 2^{56}, 2^{56}, 2^{56}, 2^{56}, 2^{56}, 2^{56}, 2^{56}, 2^{56}, 2^{56}, 2^{56}]$ WOM-code ($\mathcal{R}_{\text{sum}} = 2.0144$).

Wu, 2010 [17]

Wu designed two-write WOM-codes that had the highest sum-rates of any such WOM-codes known at the time. His best construction gave a $[10, 2; 176, 76]$ WOM-code ($\mathcal{R}_{\text{sum}} = 1.3707$). He also presented a construction of "ϵ-error" two-write WOM-codes for which the second write is not guaranteed in the worst case, but is allowed with high probability.

The results of the best previously known WOM-codes both for fixed- and unrestricted-rate WOM-codes as well as upper bounds for each case are summarized in Table 3.3.

3.4 Two-Write WOM-Codes

In this section we present a two-write WOM-codes construction that reduces the gap between the upper and lower bound on the sum-rates for both fixed- and unrestricted-rate WOM-codes. The construction is inspired by the "coset-coding" scheme which was used in [2, 5] and recently in [17]. In [2, 5], multiple-write WOM-codes are constructed where on each write the "coset-coding" scheme is used. In [17], the "coset-coding" is used only on the second write in order to generate an ϵ-error two-write WOM-codes. In ϵ-error two-write WOM-codes the second write is not guaranteed in the worst case but is allowed with high probability. Here, it is shown how to generate from every linear code a two-write WOM-code. As in [17], we use the "coset-coding" scheme only on the second write, and the first write is modified such that the second write is guaranteed in the worst case. We show two specific examples of WOM-codes having better sum-rates than the previously best known ones. We also show that by choosing uniformly at random the parity-check matrix of the linear code, there exist WOM-codes that achieve all points in the capacity region of two-write WOM-codes. Finally, we discuss the connection between the Blackwell channel [1] and two-write WOM-codes. We show how to generate from each two-write WOM-code a code for the Blackwell channel.

3.4.1 Two-Write WOM-Codes Construction

Let $\mathcal{C}[n,k]$ be a linear code with parity-check matrix $\mathcal{H}$. For each $v \in \{0,1\}^n$ we define the matrix $\mathcal{H}_v$ as follows. The i-th column of $\mathcal{H}_v$, $1 \leqslant i \leqslant n$, is the i-th column of $\mathcal{H}$ if $v_i = 0$ and otherwise it is the zeros column. The set $V_\mathcal{C}$ is defined to be

$$V_\mathcal{C} = \{v \in \{0,1\}^n \mid \text{rank}(\mathcal{H}_v) = n-k\}. \tag{3.2}$$

We first note the following claim.

Claim 3.4.1. *If a vector v belongs to $V_\mathcal{C}$, its weight is at most k.*

The support of a binary vector v, denoted by supp(v), is the set $\{i \mid v_i = 1\}$. The dual of the code $\mathcal{C}$ is denoted by $\mathcal{C}^\perp$. The next lemma is a variation of a well known result (see e.g. [2]).

Lemma 3.4.2. *Let $\mathcal{C}[n,k]$ be a linear code with parity-check matrix $\mathcal{H}$. For each vector $v \in \{0,1\}^n$, rank$(\mathcal{H}_v) = n-k$ if and only if v does not cover any non-zero codeword in $\mathcal{C}^\perp$.*

Lemma 3.4.2 implies that if two matrices are parity-check matrices of the same linear code $\mathcal{C}$, then their corresponding sets $V_\mathcal{C}$ are identical, and so we can define the set $V_\mathcal{C}$ to be

$$V_\mathcal{C} = \{v \in \{0,1\}^n \mid v \text{ does not cover any non-zero } c \in \mathcal{C}^\perp\}.$$

The next theorem presents our two-write WOM-codes.

Theorem 3.4.3. *Let $\mathcal{C}[n,k]$ be a linear code with parity-check matrix $\mathcal{H}$ and let $V_\mathcal{C}$ be the set defined in (3.2). Then there exists an $[n,2;|V_\mathcal{C}|,2^{n-k}]$ two-write WOM-code of sum-rate*

$$\frac{\log_2|V_\mathcal{C}| + (n-k)}{n}.$$

Proof. We need to show the existence of the encoding and decoding maps on the first and second writes. First, let $\{v_1, v_2, \ldots, v_{|V_{\mathcal{C}}|}\}$ be an ordering of the set $V_{\mathcal{C}}$. The first and the second writes are implemented as follows.

1. On the first write, a symbol over an alphabet of size $|V_{\mathcal{C}}|$ is written. The encoding and decoding maps $\mathcal{E}_1, \mathcal{D}_1$ are defined as follows. For each $m \in \{1, \ldots, |V_{\mathcal{C}}|\}, \mathcal{E}_1(m) = v_m$ and $\mathcal{D}_1(v_m) = m$.

2. On the second write, we write a vector $\boldsymbol{s}_2$ of $n-k$ bits. Let $\boldsymbol{v}_1$ be the programmed vector on the first write and $\boldsymbol{s}_1 = \mathcal{H} \cdot \boldsymbol{v}_1$, then

$$\mathcal{E}_2(\boldsymbol{s}_2, \boldsymbol{v}_1) = \boldsymbol{v}_1 + \boldsymbol{v}_2,$$

where $\boldsymbol{v}_2$ is a solution of the equation $\mathcal{H}_{\boldsymbol{v}_1} \cdot \boldsymbol{v}_2 = \boldsymbol{s}_1 + \boldsymbol{s}_2$. For the decoding map $\mathcal{D}_2$, if $\boldsymbol{c}$ is the vector of programmed cells, then the decoded value of the $n-k$ bits is given by

$$\mathcal{D}_2(\boldsymbol{c}) = \mathcal{H} \cdot \boldsymbol{c} = \mathcal{H} \cdot \boldsymbol{v}_1 + \mathcal{H} \cdot \boldsymbol{v}_2 = \boldsymbol{s}_1 + \boldsymbol{s}_1 + \boldsymbol{s}_2 = \boldsymbol{s}_2.$$

The success of the second write results from the condition that for every vector $\boldsymbol{v} \in V_{\mathcal{C}}$, $\text{rank}(\mathcal{H}_{\boldsymbol{v}}) = n-k$. ■

There is no condition on the code $\mathcal{C}$ and therefore we can use any linear code in this construction, though we seek to find codes that maximize the sum-rate $\frac{\log_2(|V_{\mathcal{C}}|)+n-k}{n}$. Next, we show two examples of two-write WOM-codes that achieve better sum-rates than the previously best known ones.

Example 3.4.1. Let us demonstrate how Theorem 3.4.3 works for the $[16, 5, 8]$ first order Reed-Muller code. Its dual code is the $[16, 11, 4]$ second order Reed-Muller, which is the extended Hamming code of length 16. Hence, we are interested in the size of the set

$$V_1 = \left\{ \boldsymbol{v} \in \{0,1\}^{16} | \ \boldsymbol{v} \text{ does not cover any } c \in [16, 11, 4] \right\}.$$

According to Claim 3.4.1, the set V_1 does not contain vectors of weight greater than five. This extended Hamming code has 140 codewords of weight four and no codewords of weight five. The set V_1 consists of the following vector sets.

1. All vectors of weight at most three. There are $\sum_{i=0}^{3} \binom{16}{i} = 697$ such vectors.

2. All vectors of weight four that are not codewords. There are $\binom{16}{4} - 140 = 1680$ such vectors.

3. All vectors of weight five that do no cover a codeword of weight four. There are $\binom{16}{5} - 12 \cdot 140 = 2688$ such vectors. Since the minimum distance of the code is four, a vector of weight five can cover at most one codeword of weight four.

Therefore, we get $|V_1| = 697 + 1680 + 2688 = 5065$ and the sum-rate is

$$(\log_2(5065) + 11)/16 = 1.4566.$$

It is possible to modify this WOM-code such that on the first write only 11 bits are written. Thus, we achieve a two-write fixed-rate WOM-code and its sum-rate is $22/16 = 1.375$, which is the best known fixed-rate WOM-code.

Example 3.4.2. In this example we will use the $[23, 11, 8]$ Golay code. Its dual code is the $[23, 12, 7]$ Golay code so we are interested in the size of the set

$$V_2 = \left\{ v \in \{0,1\}^{23} \mid v \text{ does not cover any } c \in [23, 12, 7] \right\}.$$

According to Claim 3.4.1, there are no vectors of weight greater than 11 in the set V_2. The $[23, 12, 7]$ Golay code has $A_7 = 253$ codewords of weight seven, $A_8 = 506$ codewords of weight eight, and $A_{11} = 1288$ codewords of weight 11. The set V_2 consists of the following vector sets.

1. All vectors of weight at most 6. This number of vectors is $\sum_{i=0}^{6} \binom{23}{i} = 145499$.

2. All vectors of weight between 7 and 10 besides those that cover a codeword of weight 7 or 8. Since the minimum distance of the code is 7 every vector can cover at most one codeword. Hence, this number of vectors is

$$\begin{aligned} &\sum_{i=7}^{10} \binom{23}{i} - A_7 \cdot \sum_{i=7}^{10} \binom{16}{i-7} - A_8 \cdot \sum_{i=8}^{10} \binom{15}{i-8} \\ &= 2459160 \end{aligned}$$

3. All vectors of weight 11 that are not codewords and do not cover a codeword of weight either 7 or 8. This number was shown in [5] to be 695520.

Therefore, for the $[23, 11, 8]$ Golay code we get

$$|V_2| = 145499 + 2459160 + 695520 = 3300179,$$

and thus the sum-rate is

$$(\log_2(3300179) + 12)/23 = 1.4632.$$

3.4.2 Random Coding

The scheme we described in the previous subsection can work for any linear code $\mathcal{C}$. Given a linear code $\mathcal{C}[n,k]$ with parity-check matrix $H_{\mathcal{C}}$, we denote $\mathcal{R}_1(\mathcal{C}) = \frac{\log_2 |V_{\mathcal{C}}|}{n}, \mathcal{R}_2(\mathcal{C}) = \frac{n-k}{n}$ so the sum-rate of the generated WOM-codes is

$$\mathcal{R}_1(\mathcal{C}) + \mathcal{R}_2(\mathcal{C}) = \frac{\log_2 |V_{\mathcal{C}}| + n - k}{n}.$$

Our goal in this subsection is to show that it is possible to achieve the capacity region C_2 defined in (3.1), by choosing uniformly at random the parity-check matrix of the linear code $\mathcal{C}$. We prove that in the following theorem.

Theorem 3.4.4. *For any $(\mathcal{R}_1, \mathcal{R}_2) \in C_2$ and $\epsilon > 0$ there exists a linear code $\mathcal{C}$ satisfying $\mathcal{R}_1(\mathcal{C}) \geqslant \mathcal{R}_1 - \epsilon, \mathcal{R}_2(\mathcal{C}) \geqslant \mathcal{R}_2 - \epsilon$.*

Proof. Let $p \in [0, 0.5]$ be such that $\mathcal{R}_1 \leqslant h(p)$ and $\mathcal{R}_2 \leqslant 1 - p$. Let $k = \lceil np \rceil$ for n large enough and let us choose uniformly at random an $(n-k) \times n$ matrix H. The matrix H will be the parity-check matrix of the linear code $\mathcal{C}$ that will be used to construct the two-write WOM-code. For each vector $v \in \{0,1\}^n$, let us define the indicator random variable $X_v(H)$ on the space of all matrices as follows

$$X_v(H) = \begin{cases} 1 & \text{if } v \in V_{\mathcal{C}} \\ 0 & \text{otherwise} \end{cases}$$

where $V_{\mathcal{C}}$ is the set defined in (3.2). Note that choosing the matrix H uniformly at random induces a probability distribution on the set $V_{\mathcal{C}}$ and thus a probability distribution on the random variable $X_v(H)$. Then the number of vectors in $V_{\mathcal{C}}$ is $X(H) = \sum_{v \in \{0,1\}^n} X_v(H)$, and

$$E[X(H)] = \sum_{v \in \{0,1\}^n} E[X_v(H)] = \sum_{v \in \{0,1\}^n} \Pr\{X_v(H) = 1\}. \tag{3.3}$$

We claim that $\Pr\{X_v(H) = 1\}$ depends on v only through its weight, $wt(v)$. In this case, (3.3) simplifies to

$$E[X(H)] = \sum_{i=0}^{n} \binom{n}{i} \Pr\{X_{v:wt(v)=i}(H) = 1\} = \sum_{i=0}^{k} \binom{n}{i} \Pr\{X_{v:wt(v)=i}(H) = 1\},$$

because if $wt(v) \geqslant k+1$ then $X_v = 0$ (Claim 3.4.1).

Now, let us determine the value of $\Pr\{X_v(H) = 1\}$ for a vector v of weight $0 \leqslant i \leqslant k$. Note that $v \in V_{\mathcal{C}}$ if and only if the sub-matrix of size $(n-k) \times (n - wt(v))$ induced by the zero entries of the vector v is full rank. It is well known, e.g. [3], that if we choose an $m \times n$ matrix, where $m \leqslant n$, uniformly at random then the probability that it is full rank is $\prod_{j=n-m+1}^{n}(1-2^{-j})$. Therefore, if we choose an $(n-k)\times(n-i)$ matrix uniformly at random then the probability that it is full rank is $\prod_{j=k-i+1}^{n-i}\left(1-2^{-j}\right)$. Note that

$$\prod_{j=k-i+1}^{n-i}\left(1-2^{-j}\right) > \prod_{j=1}^{\infty}\left(1-2^{-j}\right) > \left(1-\frac{1}{2}\right)\left(1-\sum_{j=2}^{\infty}2^{-j}\right) = \frac{1}{2}\cdot\frac{1}{2} = \frac{1}{4},$$

and, hence, $\Pr\{X_v(H)=1\} = \prod_{j=k-i+1}^{n-i}\left(1-2^{-j}\right) > 1/4$. According to Lemma 4.8 in [22],

$$\sum_{i=0}^{k}\binom{n}{i} \geqslant \frac{1}{n+1}2^{nh\left(\frac{k}{n}\right)}$$

and, therefore, we get

$$E[X(H)] = \sum_{i=0}^{k}\binom{n}{i}\prod_{j=k-i+1}^{n-i}\left(1-2^{-j}\right) > 2^{nh\left(\frac{k}{n}\right)-2-\log_2(n+1)}.$$

It follows that there exists a parity-check matrix H of a linear code $\mathcal{C}$, such that the size of the set $V_{\mathcal{C}}$ is at least $2^{nh\left(\frac{k}{n}\right)-2-\log_2(n+1)}$ and

$$\mathcal{R}_1(\mathcal{C}) \geqslant h\left(\frac{k}{n}\right) - \frac{2+\log_2(n+1)}{n} \geqslant h(p) - \frac{2+\log_2(n+1)}{n} \geqslant \mathcal{R}_1 - \epsilon$$
$$\mathcal{R}_2(\mathcal{C}) = \frac{n-k}{n} \geqslant (1-p) - \frac{1}{n} \geqslant \mathcal{R}_2 - \epsilon$$

for n large enough. ■

Random coding was proved to be capacity-achieving by constructing a partition code [6], [4]. However, our random coding scheme has more structure that enables to look for WOM-codes with a relatively small block length. We ran a computer search to look for such WOM-codes. The parity-check matrix of the linear code $\mathcal{C}$ was chosen uniformly at random and then the size of the set $V_{\mathcal{C}}$ was computed. The results are shown in Fig. 7.1. Note that if $(\mathcal{R}_1, \mathcal{R}_2)$ and $(\mathcal{R}_3, \mathcal{R}_4)$ are two achievable rate points then for each $t \in \mathbb{Q}$ the point $(t\mathcal{R}_1 + (1-t)\mathcal{R}_2, t\mathcal{R}_3 + (1-t)\mathcal{R}_4)$ is an achievable rate point, too. This can simply be done by block sharing of a large number of blocks. Therefore, the achievable region is convex.

We ran a computer search to find more two-write WOM-codes with high sum-rates. For fixed-rate WOM-codes, our best construction achieved by a computer search has sum-rate $\frac{48}{33} \approx 1.4546$ and for unrestricted-rate WOM-codes our best computer search construction achieved sum-rate 1.4928. The number of cells in these two constructions is 33.

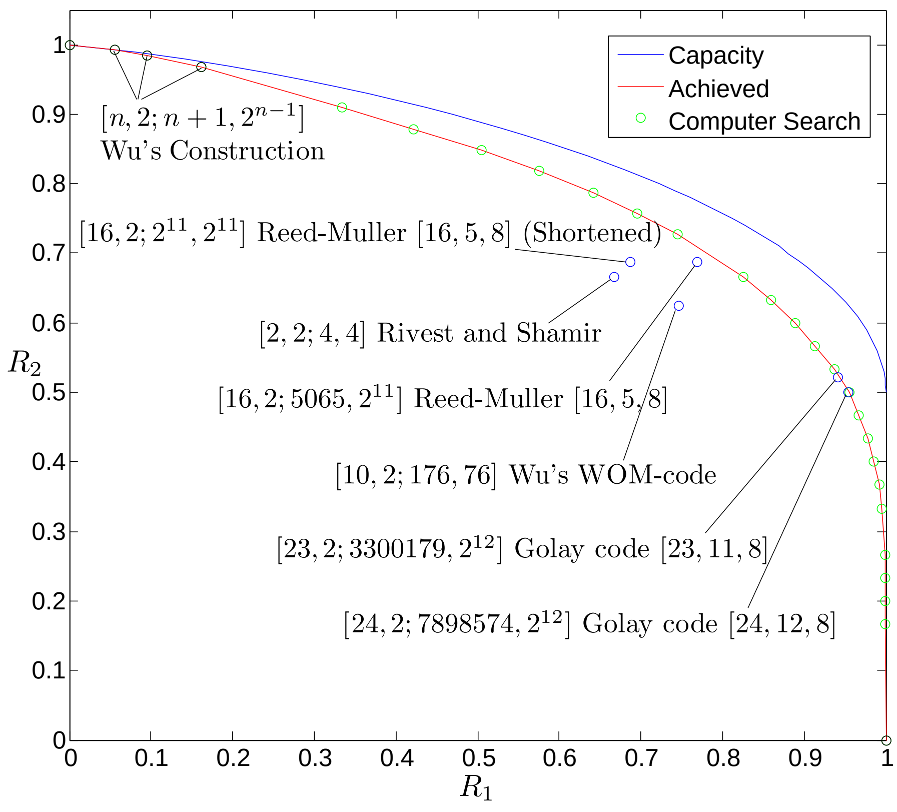


Figure 3.1: The capacity region and achieved rates of two-write WOM-codes.

Remark 3.4.1. The encoding and decoding maps of the second write are implemented by the parity-check matrix of the linear code $\mathcal{C}$ as described in the proof of Theorem 3.4.3. A naive scheme to implement the encoding and decoding maps of the first write is simply by a lookup table of the set $V_{\mathcal{C}}$. However, this can be done more efficiently using algorithms to encode and decode constant weight binary codes. There are several works which efficiently encode and decode all binary vectors of length n and weight k; see for example [2, 6, 17, 23, 24]. These works can be easily extended to construct efficient encoder and decoder maps to the set of all binary vectors of length n and weight at most k, denoted by

$$B(n,k) = \{\boldsymbol{v} \in \{0,1\}^n \mid \text{supp}(\boldsymbol{v}) \leqslant k\}.$$

The set $V_{\mathcal{C}}$ is a subset of the set $B(n,k)$. Therefore, we can use these algorithms while constructing a smaller table, only for the vectors in the set $B(n,k) \setminus V_{\mathcal{C}}$ as follows. Assume that $f : \{1,\dots,|B(n,k)|\} \to B(n,k)$ is a one-to-one and onto map such that the complexity to calculate f and f^{-1} is efficient. Assume we list all the vectors in $B(n,k) \setminus V_{\mathcal{C}}$ such that we list for every vector $\boldsymbol{v} \in B(n,k) \setminus V_{\mathcal{C}}$ its value $f^{-1}(\boldsymbol{v})$ and this list is sorted according to the values of $f^{-1}(\boldsymbol{v})$. Then, a mapping $g : \{1,\dots,|V_{\mathcal{C}}|\} \to V_{\mathcal{C}}$ is constructed such that for all $x \in \{1,\dots,|V_{\mathcal{C}}|\}$, $g(x) = f(x + a(x))$, where $a(x)$ is the number of vectors in $B(n,k) \setminus V_{\mathcal{C}}$ of value less than x. The time complexity to calculate $a(x)$ is $O(\log_2(|B(n,k) \setminus V_{\mathcal{C}}|))$ since this list is sorted. Similarly, for all $\boldsymbol{v} \in V_{\mathcal{C}}$, $g^{-1}(\boldsymbol{v}) = f^{-1}(\boldsymbol{v}) - a(f^{-1}(\boldsymbol{v}))$.

In many cases, the size of the set $B(k,n) \setminus V_{\mathcal{C}}$ will be significantly smaller than the size of $V_{\mathcal{C}}$. For example, for the Golay code $[23,11,8]$, the size of $V_{\mathcal{C}}$ is 3300179 while the size of $B(23,11) \setminus V_{\mathcal{C}}$ is

$$\sum_{i=0}^{11} \binom{n}{i} - 3300179 = 894125.$$

Similarly, for the Reed-Muller code $[16,5,8]$, the size of the set $V_{\mathcal{C}}$ is 5065 while the size of the set $B(16,5) \setminus V_{\mathcal{C}}$ is 1820.

3.4.3 Application to the Blackwell Channel

The Blackwell channel, introduced first by Blackwell [1], is one example of a deterministic broadcast channel. The channel is composed of one transmitter and two receivers. The input to the transmitter is ternary and the channel output to each receiver is a binary symbol. Let $\boldsymbol{u}$ be the ternary input vector to the transmitter of length n. For $1 \leqslant i \leqslant n$, $f(\boldsymbol{u}_i) = (f(\boldsymbol{u}_i)_1, f(\boldsymbol{u}_i)_2)$ is a binary vector of length two defined as follows (see Fig. 3.2):

$$f(0) = (0,0),\ f(1) = (0,1),\ f(2) = (1,0).$$

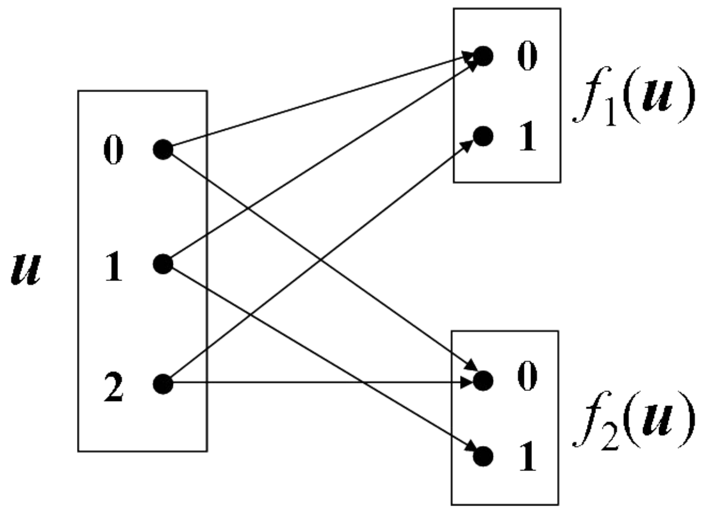


Figure 3.2: The Blackwell Channel.

The binary vectors $f_1(\boldsymbol{u}), f_2(\boldsymbol{u})$ are defined to be

$$f_1(\boldsymbol{u}) = (f(u_1)_1, f(u_2)_1, \ldots, f(u_n)_1),$$
$$f_2(\boldsymbol{u}) = (f(u_1)_2, f(u_2)_2, \ldots, f(u_n)_2),$$

and are the output vectors to the two receivers.

The capacity region of the Blackwell channel was found by Gel'fand [9] and consists of five sub-regions, given by their boundaries:

1. $\{(R_1, R_2) \mid 0 \leqslant R_1 \leqslant 1/2, R_2 = 1\}$,
2. $\{(R_1, R_2) \mid R_1 = 1 - p, R_2 = h(p), 1/3 \leqslant p \leqslant 1/2\}$,
3. $\{(R_1, R_2) \mid R_1 + R_2 = \log_2 3, \frac{2}{3} \leqslant R_1 \leqslant \log_2 3 - \frac{2}{3}\}$,
4. $\{(R_1, R_2) \mid R_1 = h(p), R_2 = 1 - p, 1/3 \leqslant p \leqslant 1/2\}$,
5. $\{(R_1, R_2) \mid R_1 = 1, 0 \leqslant R_2 \leqslant 1/2\}$.

The connection between the Blackwell channel and two-write WOM-codes was suggested by Roth [15]. The next theorem shows that from every two-write WOM-code of rate (R_1, R_2) it is possible to construct codes for the Blackwell channel of rates (R_1, R_2) and (R_2, R_1).

Theorem 3.4.5. *If (R_1, R_2) is an achievable rate of a two-write WOM-code, then (R_1, R_2) and (R_2, R_1) are achievable rates on the Blackwell channel.*

Proof. Assume that there exists a $[n, 2; 2^{nR_1}, 2^{nR_2}]$ two-write WOM-code and let $\mathcal{E}_1, \mathcal{E}_2$ and $\mathcal{D}_1, \mathcal{D}_2$ be its encoding and decoding maps. We claim that there exists a coding scheme for the Blackwell channel of rate (R_1, R_2). Let $(m_1, m_2) \in \{1, \ldots, 2^{nR_1}\} \times \{1, \ldots, 2^{nR_2}\}$ be two messages and let $\boldsymbol{v}_1 = \mathcal{E}_1(m_1)$ and $\boldsymbol{v}_2 = \mathcal{E}_2(m_2, \boldsymbol{v}_1)$. Let $\boldsymbol{u}$ be a ternary vector of length n defined as follows. For $1 \leqslant i \leqslant n$, $\boldsymbol{u}_i = f^{-1}(v_{1,i}, \overline{v_{2,i}})$. The vector $\boldsymbol{u}$ is well-defined since for all $1 \leqslant i \leqslant n$, $(v_{1,i}, v_{2,i}) \neq (1, 0)$ and hence $(v_{1,i}, \overline{v_{2,i}}) \neq (1, 1)$. The vector $\boldsymbol{u}$ is the input to the transmitter. Then, the vector $f_1(\boldsymbol{u})$ is transmitted to the first receiver and the vector $f_2(\boldsymbol{u})$ to the second receiver. Note that $f_1(\boldsymbol{u}) = \boldsymbol{v}_1$ and $f_2(\boldsymbol{u}) = \overline{\boldsymbol{v}_2}$. Therefore, the first receiver decodes its message according to $\mathcal{D}_1(f_1(\boldsymbol{u})) = \mathcal{D}_1(\boldsymbol{v}_1) = m_1$ and the second receiver decodes its message according to $\mathcal{D}_2(\overline{f_2(\boldsymbol{u})}) = \mathcal{D}_2(\boldsymbol{v}_2) = m_2$.

Similarly, it is possible to achieve the rate (R_2, R_1). Now we let $\boldsymbol{v}_2 = \mathcal{E}_2(m_2)$ and $\boldsymbol{v}_1 = \mathcal{E}_1(m_1, \boldsymbol{v}_2)$. The vector $\boldsymbol{u}$ is defined as $\boldsymbol{u}_i = f^{-1}(\overline{v_{1,i}}, v_{2,i})$ for $1 \leqslant i \leqslant n$. The decoded message by the first receiver is $\mathcal{D}_1(\overline{f_1(\boldsymbol{u})})$ and $\mathcal{D}_2(f_2(\boldsymbol{u}))$ is the decoded message by the second receiver. ■

Remark 3.4.2. It is possible to define the Blackwell channel differently such that the forbidden pair of bits is not $(1, 1)$ but another combination. Our construction of the codes can be adjusted accordingly.

Now, we can use our two-write WOM-codes in order to define codes for the Blackwell channel. By using time sharing, the achievable region is convex and hence we get in Fig. 3.3 the capacity and achieved regions for the Blackwell channel.

3.5 Multiple-Write WOM-Codes

In this section, we present WOM-code constructions which reduce the gaps between the upper and lower bounds on the sum-rates of WOM-codes for $3 \leqslant t \leqslant 10$. First, we generalize the two-write WOM-code construction from Section 3.4 for non-binary cells. Then, we show how to use these non-binary two-write WOM-codes in order to construct binary multiple-write WOM-codes. We start with specific constructions for three- and four-write WOM-codes, and then show a general design approach that works for an arbitrary number of writes.

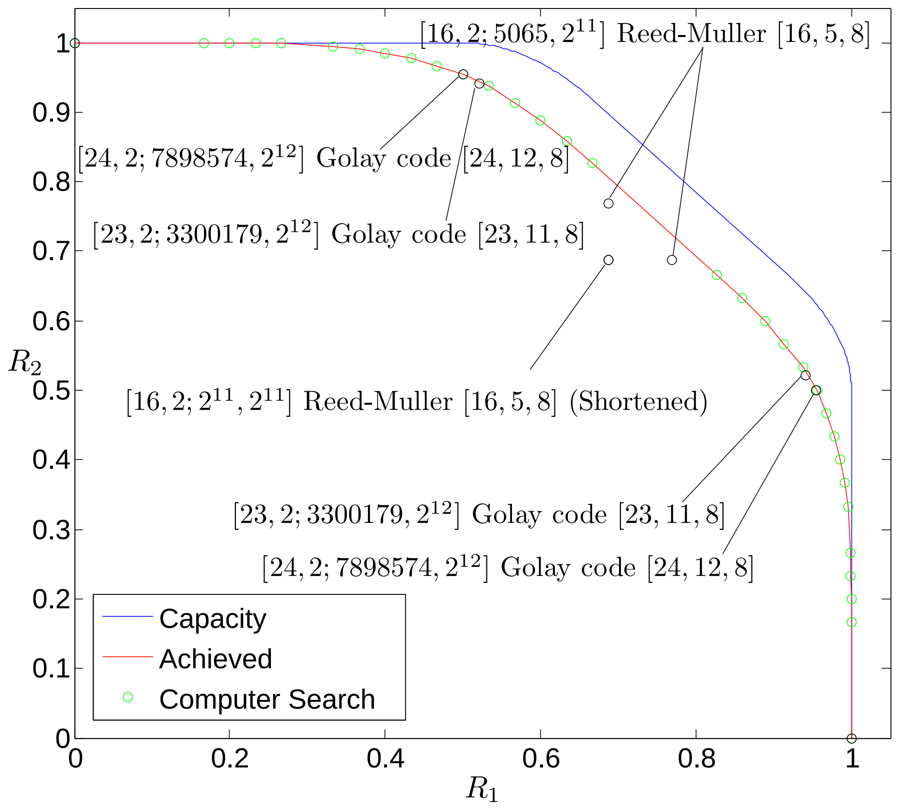


Figure 3.3: The capacity region and the achieved rates of the Blackwell channel.

3.5.1 Non-binary Two-Write WOM-Codes

Suppose now that each cell has q levels, where q is a prime number or a power of a prime number. We start by choosing a linear code $\mathcal{C}[n,k]$ over $\mathrm{GF}(q)$ with a parity-check matrix $\mathcal{H}$ of size $(n-k)\times n$. For a vector v of length n over $\mathrm{GF}(q)$, let $\mathcal{H}(v)$ be the matrix $\mathcal{H}$ with zero columns replacing the columns that correspond to the positions of the non-zero values in v. Then we define

$$V_{\mathcal{C}}^{(q)} = \{v \in (\mathrm{GF}(q))^n \mid \mathrm{rank}(\mathcal{H}(v)) = n-k\}. \tag{3.4}$$

Next, we construct a non-binary two-write WOM-code $[n,2;|V_{\mathcal{C}}^{(q)}|,q^{n-k}]$ in a similar manner to the construction in Section 3.4. Since the proof of the next theorem is very similar to the proof of Theorem 3.4.3 we omit it. A complete proof can be found in [11].

Theorem 3.5.1. *Let $\mathcal{C}[n,k]$ be a linear code with parity-check matrix $\mathcal{H}$ over $\mathrm{GF}(q)$ and let $V_{\mathcal{C}}^{(q)}$ be the set defined in (3.4). Then there exists a q-ary $[n,2;|V_{\mathcal{C}}^{(q)}|,q^{n-k}]$ two-write WOM-code of sum-rate*

$$\frac{\log_2|V_{\mathcal{C}}^{(q)}| + (n-k)\log_2 q}{n}.$$

As was shown in the binary case, there is no restriction on the choice of the linear code $\mathcal{C}$ or the parity-check matrix $\mathcal{H}$. Every such code/matrix generates a WOM-code. For a linear code $\mathcal{C}$ we define $\mathcal{R}_1(\mathcal{C}) = \frac{\log_2|V_{\mathcal{C}}^{(q)}|}{n}$ and $\mathcal{R}_2(\mathcal{C}) = \frac{(n-k)\log_2 q}{n}$ so the sum-rate of the generated WOM-code is $\mathcal{R}_1(\mathcal{C})+\mathcal{R}_2(\mathcal{C})$. The capacity region of the achievable rates by this construction is

$$C_2^{(q)} = \Big\{(\mathcal{R}_1,\mathcal{R}_2) \mid \exists p \in [0,\frac{q-1}{q}], \mathcal{R}_1 \leqslant h(p) + p\log_2(q-1), \mathcal{R}_2 \leqslant (1-p)\log_2(q)\Big\}.$$

The proof is also very similar to Theorem 3.4.4 in Section 3.4 for the binary case and thus we omit it as the complete proof appears in [11].

Theorem 3.5.2. *For any $(\mathcal{R}_1,\mathcal{R}_2) \in C_2^{(q)}$ and $\epsilon > 0$, there exists a linear code $\mathcal{C}$ satisfying $\mathcal{R}_1(\mathcal{C}) \geqslant \mathcal{R}_1 - \epsilon, \mathcal{R}_2(\mathcal{C}) \geqslant \mathcal{R}_2 - \epsilon$.*

The next corollary provides the best achievable sum-rate of the construction.

Corollary 3.5.3. *For any q-ary WOM-code generated using our construction, the highest achievable sum-rate is $\log_2(2q-1)$.*

Proof. First, note that

$$\begin{aligned}&h(p) + p\log_2(q-1) + (1-p)\log_2 q\\&= p\log_2\left(\frac{q-1}{p}\right) + (1-p)\log_2\left(\frac{q}{1-p}\right),\end{aligned}$$

and since the function $f(x) = \log_2 x$ is a concave function

$$\begin{aligned}&p\log_2\frac{q-1}{p} + (1-p)\log_2\frac{q}{1-p}\\&\leqslant \log_2\left(p\cdot\frac{q-1}{p} + (1-p)\frac{q}{1-p}\right) = \log_2(2q-1).\end{aligned}$$

Also, for $p = \frac{q-1}{2q-1}$, the achievable sum-rate is $\log_2(2q-1)$. Therefore, there exists a WOM-code produced by our construction with sum-rate $\log_2(2q-1)$.

On the other hand, any WOM-code resulting from our construction satisfies the property that every cell is programmed at most once. This model was studied in [4] and the maximum achievable sum-rate was proved to be $\log_2(2q-1)$. Therefore, our construction cannot produce a WOM-code with a sum-rate that exceeds $\log_2(2q-1)$. ■

Remark 3.5.1. This construction does not achieve high sum-rates for non-binary two-write WOM-codes in general. While the best achievable sum-rate of the construction is $\log_2(2q-1)$, the upper bound on the sum-rate is $\log_2\binom{q+1}{2}$; see [4]. The decrease in the sum-rate in our construction results from the fact that cells cannot be programmed twice. That is, if a cell was programmed on the first write, it cannot be reprogrammed on the second write even if it did not reach its highest level. In fact, it is possible to find non-binary two-write WOM-codes with better sum-rates. However, our goal in this work is not to find efficient non-binary WOM-codes. Rather, as shown later, the non-binary codes that we have constructed can be used in the design of binary multiple-write WOM-codes.

For the construction of binary multiple-write in the next subsection, we use WOM-codes over $\mathrm{GF}(3)$. We ran a computer search to find such a ternary two-write WOM-code of sum-rate 2.2205, and we will use this WOM-code in order to construct specific multiple-write WOM-codes.

3.5.2 Three-Write WOM-Codes

We start with a construction for binary three-write WOM-codes. The construction uses the WOM-codes found in the previous subsection over $\mathrm{GF}(3)$.

Theorem 3.5.4. *Let $\mathcal{C}_3$ be an $[n,2;2^{n\mathcal{R}_1},2^{n\mathcal{R}_2}]$ two-write WOM-code over* $\mathrm{GF}(3)$ *constructed as in Section* 3.5.1. *Then, there exists a $[2n,3;2^{n\mathcal{R}_1},2^{n\mathcal{R}_2},2^n]$ three-write WOM-code of sum-rate $\frac{\mathcal{R}_1+\mathcal{R}_2+1}{2}$.*

Proof. We denote by $\mathcal{E}_{3,1}$ and $\mathcal{E}_{3,2}$ the encoding maps of the first and second writes, and by $\mathcal{D}_{3,1}$ and $\mathcal{D}_{3,2}$ the decoding maps of the first and second writes of the WOM-code $\mathcal{C}_3$, respectively. The $2n$ cells of the three-write WOM-code we construct are divided into n two-cell blocks, so the memory-state vector is of the form $((c_{1,1},c_{1,2}),(c_{2,1},c_{2,2}),\ldots,(c_{n,1},c_{n,2}))$. In this construction we also use a map $\phi:\mathrm{GF}(3)\mapsto(\mathrm{GF}(2),\mathrm{GF}(2))$ defined as follows:

$$\phi(0)=(0,0),$$
$$\phi(1)=(1,0),$$
$$\phi(2)=(0,1).$$

The map ϕ extends naturally to ternary vectors $\boldsymbol{v}=(v_1,\ldots,v_n)\in\mathrm{GF}(3)^n$ using the rule

$$\phi(\boldsymbol{v})=(\phi(v_1),\ldots,\phi(v_n)).$$

On the pairs (c,c') in the image of ϕ, we define $\phi^{-1}(c,c')$ to indicate the inverse function. The map ϕ^{-1} is extended similarly to work over vectors of such bit pairs. We are now ready to describe the encoding and decoding maps for a three-write WOM-code.

1. On the first write, a message m from the set $\{1,\ldots,2^{n\mathcal{R}_1}\}$ is written in the $2n$ cells:

$$\mathcal{E}_1(m)=\phi(\mathcal{E}_{3,1}(m)).$$

   The decoding map is defined similarly, where $\boldsymbol{c}$ is the memory-state vector:

$$\mathcal{D}_1(\boldsymbol{c})=\mathcal{D}_{3,1}(\phi^{-1}(\boldsymbol{c})).$$

2. On the second write, a message m from the set $\{1,\ldots,2^{n\mathcal{R}_2}\}$ is written in the $2n$ cells as follows. Let $\boldsymbol{c}$ be the programmed vector on the first write. Then,

$$\mathcal{E}_2(m,\boldsymbol{c})=\phi(\mathcal{E}_{3,2}(m,\phi^{-1}(\boldsymbol{c}))).$$

   That is, first the memory-state vector $\boldsymbol{c}$ is converted to a ternary vector. Then, it is encoded using the encoding $\mathcal{E}_{3,2}$ and the new message, producing a new ternary memory-state vector. Finally, the last vector is converted to a $2n$-bit vector. The decoding map is defined as on the first write:

$$\mathcal{D}_2(\boldsymbol{c})=\mathcal{D}_{3,2}(\phi^{-1}(\boldsymbol{c})).$$

According to the construction of the WOM-code $\mathcal{C}_3$, no ternary cell is programmed twice and therefore each of the n pairs of bits is programmed at most once.

3. On the third write, an n-bit vector $\boldsymbol{v}$ is written. Let $\boldsymbol{c} = ((c_{1,1}, c_{1,2}), \ldots, (c_{n,1}, c_{n,2}))$ be the current memory-state vector. Then,

$$\mathcal{E}_3(\boldsymbol{v}, \boldsymbol{c}) = ((c'_{1,1}, c'_{1,2}), \ldots, (c'_{n,1}, c'_{n,2}))$$

is a vector, defined as follows. For $1 \leqslant i \leqslant n$, $(c'_{i,1}, c'_{i,2}) = (1,1)$ if $v_i = 1$ and otherwise $(c'_{i,1}, c'_{i,2}) = (c_{i,1}, c_{i,2})$. It is always possible to program the pair of bits to be $(1,1)$ since at most one cell in each pair was previously programmed. The decoding map $\mathcal{D}_2(\boldsymbol{c})$ is defined to be

$$\mathcal{D}_2(\boldsymbol{c}) = (c_{1,1} \cdot c_{1,2}, \ldots, c_{n,1} \cdot c_{n,2}).$$

That is, the decoded value of each pair of bits is one if and only if the value of both of them is one.

■

Corollary 3.5.5. *The best achievable sum-rate of a three-write WOM-code using this construction is* $(\log_2 5 + 1)/2 \approx 1.66$.

Proof. Given a two-write WOM-code $\mathcal{C}_3$ over GF(3) with rates $(\mathcal{R}_1, \mathcal{R}_2)$, the constructed binary three-write WOM-code has rates $(\mathcal{R}_1/2, \mathcal{R}_2/2, 1/2)$ and its sum-rate is $\mathcal{R} = (\mathcal{R}_1 + \mathcal{R}_2 + 1)/2$. This sum-rate is maximized when $\mathcal{R}_1 + \mathcal{R}_2$ is maximized. But $\mathcal{R}_1 + \mathcal{R}_2$ is the sum-rate of the two-write WOM-code over GF(3), which was proven in Corollary 3.5.3 to be maximized at $\log_2 5$. Then the maximum achievable sum-rate of the constructed binary three-write WOM-code is

$$\frac{\log_2 5 + 1}{2} \approx 1.66.$$

■

Using the construction of WOM-codes over GF(3) presented in the previous subsection, we can construct a three-write WOM-code of sum-rate $(2.2205 + 1)/2 = 1.6102$.

3.5.3 Four-Write WOM-Codes

We next present a construction for four-write binary WOM-codes.

Theorem 3.5.6. *Let $\mathcal{C}_3$ be an $[n, 2; 2^{n\mathcal{R}_{3,1}}, 2^{n\mathcal{R}_{3,2}}]$ two-write WOM-code over* GF(3) *which is constructed in Section* 3.5.1. *Let $\mathcal{C}_2$ be an $[n, 2; 2^{n\mathcal{R}_{2,1}}, 2^{n\mathcal{R}_{2,2}}]$ binary two-write WOM-code. Then, there exists a $[2n, 4; 2^{n\mathcal{R}_{3,1}}, 2^{n\mathcal{R}_{3,2}}, 2^{n\mathcal{R}_{2,1}}, 2^{n\mathcal{R}_{2,2}}]$ four-write WOM-code of sum-rate $\frac{\mathcal{R}_{3,1}+\mathcal{R}_{3,2}+\mathcal{R}_{2,1}+\mathcal{R}_{2,2}}{2}$.*

Proof. The proof is very similar to the one used for three-write WOM-codes. We denote by $\mathcal{E}_{3,1}, \mathcal{E}_{3,2}$ the encoding maps of the first and second writes, and by $\mathcal{D}_{3,1}, \mathcal{D}_{3,2}$ the decoding maps of the first and second writes of the WOM-code $\mathcal{C}_3$, respectively. Similarly, the encoding and decoding maps of the WOM-code $\mathcal{C}_2$ for the first and second writes are denoted by $\mathcal{E}_{2,1}, \mathcal{E}_{2,2}$ and $\mathcal{D}_{2,1}, \mathcal{D}_{2,2}$, respectively. Using the encoding and decoding maps of $\mathcal{C}_3$, we define the first and second writes of our constructed four-write WOM-code as we did for the first and second writes of the three-write WOM-codes. The third and fourth writes are defined in a similar way, as follows.

1. On the third write, a message m from the set $\{1, \ldots, 2^{n\mathcal{R}_{2,1}}\}$ is written. Let $\mathcal{E}_{2,1}(m) = \boldsymbol{v} = (v_1, \ldots, v_n)$ and let $\boldsymbol{c} = ((c_{1,1}, c_{1,2}), \ldots, (c_{n,1}, c_{n,2}))$ be the current memory-state vector. Then,

$$\mathcal{E}_3(m, \boldsymbol{c}) = ((c'_{1,1}, c'_{1,2}), \ldots, (c'_{n,1}, c'_{n,2})),$$

   where for $1 \leqslant i \leqslant n$, $(c'_{i,1}, c'_{i,2}) = (1, 1)$ if $v_i = 1$ and, otherwise, $(c'_{i,1}, c'_{i,2}) = (c_{i,1}, c_{i,2})$. The decoding map $\mathcal{D}_3(\boldsymbol{c})$ is defined to be

$$\mathcal{D}_3(\boldsymbol{c}) = \mathcal{D}_{2,1}(c_{1,1} \cdot c_{1,2}, \ldots, c_{n,1} \cdot c_{n,2}).$$

2. On the fourth write, a message m from the set $\{1, \ldots, 2^{n\mathcal{R}_{2,2}}\}$ is written. Let

$$\mathcal{E}_{2,2}(m, (c_{1,1} \cdot c_{1,2}, \ldots, c_{n,1} \cdot c_{n,2})) = \boldsymbol{v} = (v_1, \ldots, v_n),$$

   where $\boldsymbol{c} = ((c_{1,1}, c_{1,2}), \ldots, (c_{n,1}, c_{n,2}))$ is the current memory-state vector. Then,

$$\mathcal{E}_4(m, \boldsymbol{c}) = ((c'_{1,1}, c'_{1,2}), \ldots, (c'_{n,1}, c'_{n,2})),$$

   where for $1 \leqslant i \leqslant n$, $(c'_{i,1}, c'_{i,2}) = (1, 1)$ if $v_i = 1$ and, otherwise, $(c'_{i,1}, c'_{i,2}) = (c_{i,1}, c_{i,2})$. The decoding map $\mathcal{D}_4(\boldsymbol{c})$ is defined, as before, by

$$\mathcal{D}_4(\boldsymbol{c}) = \mathcal{D}_{2,2}(c'_{1,1} \cdot c'_{1,2}, \ldots, c'_{n,1} \cdot c'_{n,2}).$$

■

Remark 3.5.2. The last theorem requires both the binary two-write and ternary two-write WOM-codes to have the same number of cells, n. However, we can construct a four-write binary WOM-code using any two such WOM-codes, even if they do not have the same number of cells. Suppose we have a WOM-code over $\mathrm{GF}(3)$ with n_1 cells and binary WOM-code with n_2 cells. Both codes can be extended to use $\mathrm{lcm}(n_1, n_2)$ cells. Then, the construction above will give a four-write WOM-code.

Corollary 3.5.7. *The best achievable sum-rate of a four-write WOM-code using this construction is* $(\log_2 5 + \log_2 3)/2 \approx 1.95$.

Proof. According to Corollary 3.5.3, the maximum value of $\mathcal{R}_{3,1} + \mathcal{R}_{3,2}$ is $\log_2 5$ and the maximum value of $\mathcal{R}_{2,1} + \mathcal{R}_{2,2}$ is $\log_2 3$. Therefore, the maximum sum-rate of the constructed four-write WOM-codes is

$$\frac{\log_2(5) + \log_2(3)}{2} \approx 1.95.$$

■

If we use the WOM-code over $\mathrm{GF}(3)$ of sum-rate 2.2205 found in the previous subsection as the WOM-code $\mathcal{C}_3$ and the binary two-write WOM-code of sum-rate 1.4928 found in Section 3.4 as the WOM-code $\mathcal{C}_2$, then there exists a four-write WOM-code of sum-rate $(2.2205 + 1.4928)/2 = 1.8566$.

3.5.4 Multiple-Write WOM-Codes

The construction of three- and four-write WOM-codes can be easily generalized to an arbitrary number of writes. We state the following theorem and skip its proof since it is very similar to the proofs of the corresponding theorems for three- and four-write WOM-codes.

Theorem 3.5.8. *Let* $\mathcal{C}_3$ *be an* $[n, 2; 2^{n\mathcal{R}_{3,1}}, 2^{n\mathcal{R}_{3,2}}]$ *two-write WOM-code over* $\mathrm{GF}(3)$ *constructed as in Section* 3.5.1. *Let* $\mathcal{C}_2$ *be an* $[n, t-2; 2^{n\mathcal{R}_{2,1}}, \ldots, 2^{n\mathcal{R}_{2,t-2}}]$ *binary* $(t-2)$*-write WOM-code. Then, there exists a*

$$[2n, t; 2^{n\mathcal{R}_{3,1}}, 2^{n\mathcal{R}_{3,2}}, 2^{n\mathcal{R}_{2,1}}, \ldots, 2^{n\mathcal{R}_{2,t-2}}]$$

t*-write WOM-code of sum-rate*

$$\frac{\mathcal{R}_{3,1} + \mathcal{R}_{3,2} + \sum_{i=1}^{t-2} \mathcal{R}_{2,i}}{2}.$$

Theorem 3.5.8 implies that if there exists a $(t-2)$-write WOM-code of sum-rate $\mathcal{R}_{t-2}$ then there exists a t-write WOM-code of sum-rate

$$\mathcal{R}_t = \frac{\log_2 5 + \mathcal{R}_{t-2}}{2}.$$

The following corollary summarizes the possible achievable sum-rates of t-write WOM-codes.

Corollary 3.5.9. *For $t \geqslant 3$, there exists a t-write WOM-code of sum-rate*

$$\mathcal{R}_t = \begin{cases} \frac{(2^{\frac{t-1}{2}}-1)\cdot\log_2 5+1}{2^{\frac{t-1}{2}}} & , t \text{ odd} \\ \frac{(2^{\frac{t-2}{2}}-1)\cdot\log_2 5+\log_2 3}{2^{\frac{t-2}{2}}} & , t \text{ even.} \end{cases}$$

If we use again the two-write WOM-code over $\mathrm{GF}(3)$ of sum-rate 2.2205 and the binary two-write WOM-code of sum-rate 1.4928 from Section 3.4, then for $t \geqslant 3$ we obtain a t-write WOM-code of sum-rate $\mathcal{R}_t$, where

$$\mathcal{R}_t = \begin{cases} \frac{(2^{\frac{t-1}{2}}-1)\cdot 2.22+1}{2^{\frac{t-1}{2}}} & , t \text{ odd} \\ \frac{(2^{\frac{t-2}{2}}-1)\cdot 2.22+1.4928}{2^{\frac{t-2}{2}}} & , t \text{ even.} \end{cases}$$

3.6 Concatenated WOM-Codes

The construction presented in the previous section provides us with a family of WOM-codes for all $t \geqslant 3$. In this section, we will show a general scheme to construct more families of WOM-codes. In fact, the construction in the previous section is a special case of this general scheme.

Theorem 3.6.1. *Assume that $\mathcal{C}^*$ is an $[m, t/2; q_1, \ldots, q_{t/2}]$ binary $t/2$-write WOM-code where t is a positive even integer. For $1 \leqslant i \leqslant t/2$, we let $\mathcal{C}_i$ be an $[n, 2; 2^{n\mathcal{R}_{i,1}}, 2^{n\mathcal{R}_{i,2}}]$ two-write WOM-code over $\mathrm{GF}(q_i)$, which is constructed in Section 3.5.1. Then, there exists an $[mn, 2^{n\mathcal{R}_{1,1}}, 2^{n\mathcal{R}_{1,2}}, \ldots, 2^{n\mathcal{R}_{t/2,1}}, 2^{n\mathcal{R}_{t/2,2}}, t]$ binary t-write WOM-code of sum-rate*

$$\sum_{i=1}^{t/2} \frac{\mathcal{R}_{i,1} + \mathcal{R}_{i,2}}{m}.$$

Proof. For $1 \leqslant i \leqslant t/2$, let $\mathcal{E}_i^*, \mathcal{D}_i^*$ be the encoding, decoding maps on the i-th write of the WOM-code $\mathcal{C}^*$, respectively. The definition of $\mathcal{E}_i^*, \mathcal{D}_i^*$ for $1 \leqslant i \leqslant t/2$ extends naturally to vectors by simply invoking the maps on each entry in the vector. Similarly, for $1 \leqslant i \leqslant t/2$, let us denote by $\mathcal{E}_{i,1}$ and $\mathcal{E}_{i,2}$ the encoding maps of the first and second writes, and by $\mathcal{D}_{i,1}$

and $\mathcal{D}_{i,2}$ the decoding maps of the first and second writes of the WOM-code $\mathcal{C}_i$, respectively. We will present the specification of the encoding and decoding maps of the constructed t-write WOM-code.

In the following definitions of the encoding and decoding maps, we consider the memory-state vector $\boldsymbol{c}$ to have n symbols of m bits each, i.e. $\boldsymbol{c} \in (\mathrm{GF}(2^m))^n$. For $1 \leqslant i \leqslant t/2$, the $(2i-1)$-st write and $2i$-th write are implemented as follows.

1. On the $(2i-1)$-st write, a message $m_1 \in \{1, \ldots, 2^{n\mathcal{R}_{i,1}}\}$ is written to the memory-state vector $\boldsymbol{c}$ according to
$$\mathcal{E}_{2i-1}(m_1, \boldsymbol{c}) = \mathcal{E}_i^*(\mathcal{E}_{i,1}(m_1), \boldsymbol{c}).$$
The memory-state vector $\boldsymbol{c}$ is decoded according to
$$\mathcal{D}_{2i-1}(\boldsymbol{c}) = \mathcal{D}_{i,1}(\mathcal{D}_i^*(\boldsymbol{c})).$$

2. On the $2i$-th write, a message $m_2 \in \{1, \ldots, 2^{n\mathcal{R}_{i,2}}\}$ is written according to
$$\mathcal{E}_{2i}(m_2) = \mathcal{E}_i^*(\mathcal{E}_{i,2}(m_2, \mathcal{D}_i^*(\boldsymbol{c})), \boldsymbol{c})$$
and the memory-state vector $\boldsymbol{c}$ is decoded according to
$$\mathcal{D}_{2i}(\boldsymbol{c}) = \mathcal{D}_{i,2}(\mathcal{D}_i^*(\boldsymbol{c})).$$

■

We will demonstrate how this construction works in the following example.

Example 3.6.1. We choose a $[3,3;4,3,2]$ three-write WOM-code as the code $\mathcal{C}^*$. This code is depicted in Fig. 3.4 by a state diagram describing all three writes. The three-bit vector in each state is the memory-state and the number next to it is the decoded value. We need to find three more two-write WOM-codes over $\mathrm{GF}(4), \mathrm{GF}(3)$, and $\mathrm{GF}(2)$. For the code $\mathcal{C}_1$ over $\mathrm{GF}(4)$, we ran a computer search to find a two-write WOM-code over $\mathrm{GF}(4)$ of sum-rate 2.6862. For the code $\mathcal{C}_2$ over $\mathrm{GF}(3)$, we use the code of sum-rate 2.22 which we found in Section 3.5.1, and we use the binary two-write WOM-code with sum-rate 1.4928 for the code $\mathcal{C}_3$. Then, the sum-rate of the six-write WOM-code is
$$\frac{2.6793 + 2.22 + 1.49}{3} = 2.1297.$$

Table 3.4: Sum-rates of concatenated WOM-codes

Number of Writes	Achieved New Rate	Maximum New Rate
5	1.9689	$\frac{\log_2 7+\log_2 5+1}{3} = 2.0431$
6	2.1331	$\frac{\log_2 7+\log_2 5+\log_2 3}{3} = 2.2381$
7	2.1723	$\frac{\log_2 7+\log_2 5+(\log_2 5+1)/2}{3} = 2.2634$
8	2.2544	$\frac{\log_2 7+\log_2 5+(\log_2 5+\log_2 3)/2}{3} = 2.3609$
9	2.2918	$\frac{\log_2 7+\log_2 5+(\log_2 7+\log_2 5+1)/3}{3} = 2.3908$
10	2.3466	$\frac{\log_2 7+\log_2 5+(\log_2 7+\log_2 5+\log_2 3)/3}{3} = 2.4588$

It is possible to construct a five-write WOM-code by writing a vector of n bits in the last write so its sum-rate is

$$\frac{2.6862+2.2205+1}{3} = 1.9689.$$

Note that if one of the codes in the general construction is binary then we can actually use a WOM-code that allows more than two writes. That is, in this construction we can use any binary multiple-write WOM-code as the WOM-code $\mathcal{C}_3$. Therefore, we can generate another family of WOM-codes for $t \geqslant 5$. Their maximum achievable sum-rates are given by the following formula

$$\mathcal{R}_t = \frac{\log_2 7+\log_2 5+\mathcal{R}_{t-4}}{3},$$

for $t \geqslant 5$ and $\mathcal{R}_{t-4}$ is the maximum achievable sum-rate for a $(t-4)$-write WOM-code. Similarly, the constructed codes which we obtain using the WOM-codes found above have sum-rates

$$\mathcal{R}'_t = \frac{2.6862+2.2205+\mathcal{R}'_{t-4}}{3},$$

where $\mathcal{R}'_{t-4}$ is the best sum-rate of a constructed $(t-4)$-write WOM-code. Table 3.4 summarizes these sum-rates.

Note that the construction in Section 3.5 is a special case of the generalized concatenated WOM-codes construction in which the WOM-code $\mathcal{C}^*$ is chosen to be a $[2,2;3,2]$ binary two-write WOM-code.

The general scheme described in Theorem 3.6.1 provides us with many more families of WOM-codes. However, in order to construct WOM-codes with high sum-rates, the WOM-code $\mathcal{C}^*$ has to be chosen very carefully. In particular, it is important to choose such a WOM-code with as few cells as possible, since the sum of all sum-rates of the non-binary two-write WOM-codes is averaged over the number of cells of the WOM-code $\mathcal{C}^*$. As the number of short WOM-codes is small, there are only a small number of possibilities to check. However, our search for better

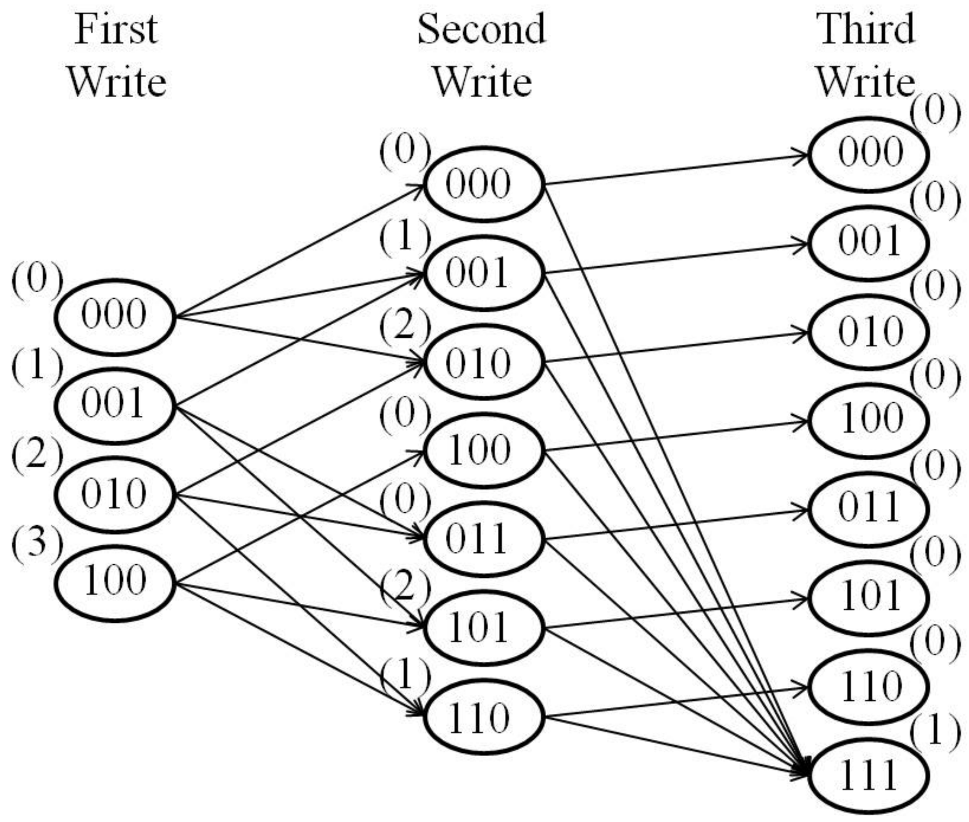


Figure 3.4: A $[3, 3; 4, 3, 2]$ three-write WOM-code.

WOM-codes with between six and ten writes using WOM-codes with few cells did not lead to any better results.

3.7 Fixed-rate WOM-codes

The WOM-code construction for more than two writes improved the achieved sum-rates only in the unrestricted-rate case. In this section, we present a method to construct fixed-rate WOM-codes. The method is recursive and is based on the previously constructed unrestricted-rate WOM-codes.

Theorem 3.7.1. *Let $\mathcal{C}$ be an $[n, t; 2^{n\mathcal{R}_1}, 2^{n\mathcal{R}_2}, \ldots, 2^{n\mathcal{R}_t}]$ t-write WOM-code. Assume that for $1 \leqslant i \leqslant t-1$ there exists a fixed-rate WOM-code of sum-rate R_i. Let $\mathcal{R}'_1, \ldots, \mathcal{R}'_t$ be a permutation of $\mathcal{R}_1, \ldots, \mathcal{R}_t$ such that $\mathcal{R}'_1 \geqslant \cdots \geqslant \mathcal{R}'_t$. Then, there exists a fixed-rate t-write WOM-code of sum-rate*

$$\frac{t \cdot \mathcal{R}'_1}{1 + \sum_{i=1}^{t-1} \frac{i(\mathcal{R}'_{t-i} - \mathcal{R}'_{t-i+1})}{R_i}}.$$

Proof. For simplicity, let us assume that $\mathcal{R}_1 \geqslant \cdots \geqslant \mathcal{R}_t$ as it will be clear from the proof how to generalize to the arbitrary case. First, we add $(\mathcal{R}_{t-1} - \mathcal{R}_t)n$ more cells in order to write $(\mathcal{R}_{t-1} - \mathcal{R}_t)n$ bits on the last write. This guarantees that the rates on the last two writes are the same. Then, we add $2(\mathcal{R}_{t-2} - \mathcal{R}_{t-1})n/R_2$ more cells in order to write $(\mathcal{R}_{t-2} - \mathcal{R}_{t-1})n$ more bits on each of the last two writes. This part of the last two writes is invoked using the fixed-rate two-write WOM-code of sum-rate R_2 and therefore the additional number of cells is $2(\mathcal{R}_{t-2} - \mathcal{R}_{t-1})n/R_2$. This addition of cells guarantees that the rates on the last three writes are all the same. In general, for $1 \leqslant i \leqslant t-1$ we add $i(\mathcal{R}_{t-i} - \mathcal{R}_{t-i+1})n/R_i$ more cells such that $(\mathcal{R}_{t-i} - \mathcal{R}_{t-i+1})n$ more bits are written on each of the last i writes and therefore the rates on the last $i+1$ writes are all the same. These bits are written using the fixed-rate i-write WOM-code which is assumed to exist.

With the addition of these cells, the number of bits written on the i-th write for $1 \leqslant i \leqslant t$ is

$$\mathcal{R}_i n + \sum_{j=1}^{i-1} (\mathcal{R}_j - \mathcal{R}_{j+1})n = \mathcal{R}_1 n.$$

Thus, the rates on all writes are the same and the generated WOM-code is fixed-rate.

The total number of bits we add is

$$\sum_{i=1}^{t-1} \frac{i(\mathcal{R}_{t-i} - \mathcal{R}_{t-i+1})n}{R_i},$$

and thus the sum-rate is

$$\frac{t \cdot \mathcal{R}_1 n}{n + \sum_{i=1}^{t-1} \frac{i(\mathcal{R}_{t-i} - \mathcal{R}_{t-i+1})n}{R_i}} = \frac{t \cdot \mathcal{R}_1}{1 + \sum_{i=1}^{t-1} \frac{i(\mathcal{R}_{t-i} - \mathcal{R}_{t-i+1})}{R_i}}.$$

■

Let us demonstrate how to apply the last theorem. We start with the three-write WOM-code we constructed in Section 3.5.2. Its rates on the first, second, and third writes are 0.6291, $0.4811, 0.5$, respectively. We add $0.0189n$ more cells in order to guarantee that the rates on the last two writes are the same. Then we use the fixed-rate two-write WOM-code constructed in Section 3.4.1 of sum-rate 1.4546. Hence we add

$$\frac{2 \cdot (0.6291 - 0.5)n}{1.4546} = 0.1775n$$

more cells, yielding a fixed-rate three-write WOM-code of sum-rate

$$\frac{3 \cdot 0.6291}{1.1964} = 1.5775.$$

If we used the best fixed-rate two-write WOM-code of sum-rate 1.546 and the best three-write WOM-code of sum-rate 1.66, then we get a fixed-rate three-write WOM-code of sum-rate 1.6263.

Note that we could use a two-write WOM-code such that $0.0189n$ bits are written on its first write and $0.1291n$ bits are written on its second write. This will indeed add another small improvement to the sum-rate, however this scheme is not easy to generalize. Our goal here is to give a general scheme. We are aware that for each individual case it is possible to use other unrestricted-rate WOM-codes that will provide a WOM-code of the desired sum-rate with slightly fewer cells.

Now we move to the four-write WOM-code from Section 3.5.3. Its component rates are $0.6291, 0.4811, 0.413, 1/3$. We add three more groups of cells as follows:

1. $(0.413 - 1/3)n = 0.0797n$ more cells, so that the last two write have the same rate.

2. $2 \cdot (0.4811 - 0.413)n/1.4546 = 0.0936n$ more cells, so that the last three writes have the same rate.

3. $3 \cdot (0.6291 - 0.4811)n/1.5731 = 0.2822n$ more cells, so that the last four writes have the same rate.

Table 3.5: Sum-rates of fixed-rate WOM-codes

Number of Writes	Achieved Sum-rate	Maximum Sum-rate
3	1.5775	1.6263
4	1.7298	1.8249
5	1.8794	1.9302
6	1.9742	2.0570
7	1.991	2.0692
8	2.0375	2.1190
9	2.0951	2.1702
10	2.1327	2.2189

Then, we get a fixed-rate four-write WOM-code with sum-rate

$$\frac{4 \cdot 0.6291}{1 + 0.0797 + 0.0936 + 0.2822} = 1.7298.$$

If we used the best fixed-rate two- and three-write WOM-codes and the best unrestricted-rate four-write WOM-code, then we obtain a fixed-rate four-write WOM-code of sum-rate 1.8249. Fixed-rate t-write WOM-code for $t > 4$ can be similarly obtained. We summarize the results for the sum-rates that we actually found and the best ones we could find in this method in Table 3.5.

3.8 Summary and Comparison

In this chapter, we have presented several constructions for multiple-write WOM-codes. First, we showed a method to construct two-write WOM-codes. Using this method we found two-write WOM-codes with better sum-rates than the previously known codes. Then, we proved that it is possible to achieve each point in the achieved-rate region of two-write WOM-codes using this scheme. Furthermore, we showed that each two-write WOM-code generates a code for the Blackwell channel.

We then presented another method for constructing binary multiple-write WOM-codes. The method made use of two-write WOM-codes over $\mathrm{GF}(q)$, for which we generalized the binary construction. While the non-binary WOM-codes we constructed do not achieve high sum-rate, they allowed us to construct binary t-write WOM-codes for $t \geqslant 3$. We showed how to construct WOM-codes for three and four writes, and then showed that a recursive algorithm can be used to generate binary WOM-codes that support any number of writes. We also described a general concatenation scheme to construct other families of WOM-codes. Applying this scheme,

Table 3.6: Comparison with known unrestricted-rate WOM-codes

Number of Writes	Best Prior	Achieved New Sum-rate	Maximum New Sum-rate	Upper Bound
2	1.3707	**1.4928**	1.585	1.585
3	1.5302	**1.6102**	1.661	2
4	1.7524	**1.8566**	1.9534	2.3219
5	1.7524	**1.9689**	2.0431	2.585
6	1.7524	**2.1331**	2.2381	2.8074
7	1.8232	**2.1723**	2.2634	3
8	1.8824	**2.2544**	2.3609	3.1699
9	1.9535	**2.2918**	2.3908	3.3219
10	2.0144	**2.3466**	2.4588	3.4594

we found another family of t-write WOM-codes that gives the best known unrestricted-rate sum-rates for $5 \leqslant t \leqslant 10$. Lastly, we showed two methods to construct fixed-rate multiple-write WOM-codes.

Tables 3.6 and 3.7 show a comparison of the sum-rates of unrestricted-rate and fixed-rate WOM-codes presented in this work and the best previously known sum-rates for $2 \leqslant t \leqslant 10$. The column labeled "Best Prior" is the highest sum-rate achieved by a previously reported t-write WOM-code. The column "Achieved New Sum-rate" gives the sum-rates that we actually obtained through application of the new techniques. The column "Maximum New Sum-rate" lists the maximum possible sum-rates that can be obtained using our approach. Finally, the column "Upper Bound" gives the maximum possible sum-rates for t-write WOM codes.

For unrestricted-rate two-write WOM-codes, the results were found by the computer search method of Section 3.4. For three and four writes, we used the WOM-codes described in Section 3.5, and for $5 \leqslant t \leqslant 10$, we used the WOM-codes discussed in Section 3.6. For fixed-rate two-write WOM-codes, we again used the computer search method of Section 3.4. The constructions for more than two writes were obtained by application of Theorem 3.7.1.

Acknowledgment

This chapter is in part a reprint of the material in the papers: E. Yaakobi, S. Kayser, P.H. Siegel, A. Vardy, and J.K. Wolf, "Efficient Two-Write WOM-Codes," *Proc. IEEE Information Theory Workshop*, Dublin, Ireland, August-September 2010, and S. Kayser, E. Yaakobi, P.H. Siegel, A. Vardy, and J.K. Wolf, "Multiple-Write WOM-Codes," *48-th Annual Allerton Conference on Communication, Control and Computing*, Monticello, IL, September 2010.

Table 3.7: Comparison with known fixed-rate WOM-codes

Number of Writes	Best Prior	Achieved New Sum-rate	Maximum New Sum-rate	Upper Bound
2	1.343	**1.4546**	1.546	1.546
3	1.4348	**1.5775**	1.6263	1.9366
4	1.6042	**1.7298**	1.8249	2.2436
5	1.6279	**1.8794**	1.9302	2.4965
6	1.7143	**1.9742**	2.0570	2.7120
7	1.8232	**1.991**	2.0692	2.9001
8	1.8824	**2.0375**	2.1190	3.0664
9	1.9535	**2.0951**	2.1702	3.2157
10	2.0144	**2.1327**	2.2189	3.3520

Bibliography

[1] D. Blackwell, “Statistics 262,” Course taught at the University of California, Berkeley, Spring 1963.

[2] T. Berger, F. Jelinek, and J.K. Wolf, “Permutation codes for sources,” *IEEE Trans. Inform. Theory*, vol. 18, no. 1, pp. 160–168, January 1972.

[3] R. Brent, S. Gao, and A. Lauder, “Random Krylov spaces over finite fields,” *SIAM J. Discrete Math*, vol. 16, no. 2, pp. 276–287, February 2003.

[4] G.D. Cohen, P. Godlewski, and F. Merkx, “Linear binary code for write-once memories,” *IEEE Trans. Inform. Theory*, vol. 32, no. 5, pp. 697–700, October 1986.

[5] J.H. Conway and N.J. Sloane, “Orbit and coset analysis of the Golay and related codes,” *IEEE Trans. Inform. Theory*, vol. 36, no. 5, pp. 1038–1050, September 1990.

[6] T.M. Cover, “Enumerative source encoding,” *IEEE Trans. Inform. Theory*, vol. 19, no. 1, pp 73–77, January 1973.

[7] A. Fiat and A. Shamir, “Generalized “write-once” memories,” *IEEE Trans. Inform. Theory*, vol. 30, no. 3, pp. 470–480, September 1984.

[8] F. Fu and A.J. Han Vinck, “On the capacity of generalized write-once memory with state transitions described by an arbitrary directed acyclic graph,” *IEEE Trans. Inform. Theory*, vol. 45, no. 1, pp. 308–313, September 1999.

[9] S.I. Gel’fand, “Capacity of one broadcast channel,” *Problemy Peredachi Informatsii*, vol. 13, no. (3), pp. 106–108, 1977.

[10] P. Godlewski, “WOM-codes construits à partir des codes de Hamming,” *Discrete Math.*, vol. 65, no. 3, pp. 237–243, July 1987.

[11] L. Grupp, A. Caulfield, J. Coburn, S. Swanson, E. Yaakobi, P.H. Siegel, and J.K. Wolf, “Characterizing flash memory: anomalies, observations, and applications,” *MICRO-42*, pp. 24–33, December 2009.

[12] C. Heegard, "On the capacity of permanent memory," *IEEE Trans. Inform. Theory*, vol. 31, no. 1, pp. 34–42, January 1985.

[13] A. Jiang, "On the generalization of error-correcting WOM codes," *Proc. IEEE Int. Symp. Inform. Theory*, pp. 1391–1395, Nice, France, June 2007.

[14] A. Jiang, V. Bohossian, and J. Bruck, "Floating codes for joint information storage in write asymmetric memories," *Proc. IEEE Int. Symp. Inform. Theory*, pp. 1166–1170, Nice, France, June 2007.

[15] A. Jiang and J. Bruck, "Joint coding for flash memory storage," *Proc. IEEE Int. Symp. Inform. Theory*, pp. 1741–1745, Toronto, Canada, July 2008.

[16] S. Kayser, E. Yaakobi, P.H. Siegel, A. Vardy, and J.K. Wolf, "Multiple-write WOM-codes," *Proc. 48-th Annual Allerton Conference on Communication, Control and Computing*, Monticello, IL, September 2010.

[17] D.E. Knuth, "Efficient balanced codes," *IEEE Trans. Inform. Theory*, vol. 32, no. 1, pp. 51–53, January 1986.

[18] H. Mahdavifar, P.H. Siegel, A. Vardy, J.K. Wolf, and E. Yaakobi, "A nearly optimal construction of flash codes," *Proc. IEEE Int. Symp. Inform. Theory*, pp. 1239–1243, Seoul, Korea, July 2009.

[19] F. Merkx, "Womcodes constructed with projective geometries," *Traitement du signal*, vol. 1, no. 2-2, pp. 227–231, 1984.

[20] R.L. Rivest and A. Shamir, "How to reuse a write-once memory," *Inform. and Contr.*, vol. 55, no. 1–3, pp. 1–19, December 1982.

[21] R.M. Roth, February 2010, personal communication.

[22] R.M. Roth, *Introduction to Coding Theory*, Cambridge University Press, 2005.

[23] J.P.M. Schalkwijk, "An algorithm for source coding," *IEEE Trans. Inform. Theory*, vol. 18, no. 3, pp. 395–399, May 1972.

[24] C. Tian, V.A. Vaishampayan, and N.J.A. Sloane, "Constant Weight codes: a geometric approach based on dissections,", *CiteSeerX - Scientific Literature Digital Library and Search Engine*, 2010.

[25] J.K. Wolf, A.D. Wyner, J. Ziv, and J. Korner, "Coding for a write-once memory," *AT&T Bell Labs. Tech. J.*, vol. 63, no. 6, pp. 1089–1112, 1984.

[26] Y. Wu, "Low complexity codes for writing write-once memory twice," *Proc. IEEE Int. Symp. Inform. Theory*, pp. 1928–1932, Austin, Texas, June 2010.

[27] Y. Wu and A. Jiang, "Position modulation code for rewriting write-once memories," accepted by *IEEE Trans. Inform. Theory*, October 2010.

[28] E. Yaakobi, J. Ma, L. Grupp, P.H. Siegel, S. Swanson, and J.K. Wolf, "Error characterization and coding schemes for flash memories," *Proc. Workshop on the Application of Communication Theory to Emerging Memory Technologies*, Miami, Florida, December 2010.

[29] E. Yaakobi, P.H. Siegel, A. Vardy, and J.K. Wolf, "Multiple error-correcting WOM-codes," *Proc. IEEE Int. Symp. Inform. Theory*, pp. 1933–1937, Austin, Texas, June 2010.

[30] G. Zémor, "Problèmes combinatoires liés à l'écriture sur des mémoires," Ph.D. Dissertation, ENST, Paris, France, November 1989.

[31] G. Zémor and G.D. Cohen, "Error-correcting WOM-codes," *IEEE Trans. Inform. Theory*, vol. 37, no. 3, pp. 730–734, May 1991.

Chapter 4

Multiple Error-Correcting WOM-Codes

4.1 Introduction

In Chapter 3, we studied write-once-memory (WOM) codes which were first presented by Rivest and Shamir almost three decades ago [5]. The first motivation to design WOM-codes came from storage medium such as punch cards and optical disks. These memories consist of binary memory elements that can only be changed from a zero state to a one state. Since then, more research results have appeared on this topic, e.g. [2–6, 13, 16, 20, 21], and recently, there has been renewed interest in these codes due to their high relevance with the ubiquitous flash memories [11, 17–19].

In the WOM model, the problem that has received the most attention is: *what is the minimum number of cells n required to store k bits t times*? Or, alternatively: *what is the maximum number of bits k that can be written t times using n cells*? Even though the problem of adapting WOM-codes to handle memory errors was suggested in the original Rivest-Shamir paper [5], the first construction of codes addressing this problem wasn't published until a few years later by Zémor [21] and Zémor and Cohen [20]. The capacity of a noisy WOM was studied by Heegard [6]. Recently, in [7], Jiang discussed the generalization of error-correcting WOM-codes for the flash/floating codes model [8, 9, 12].

4.2 Preliminaries and Previous Work

In this chapter, the memory elements, called ***cells***, have two states: zero and one. At the beginning, all the cells are in their zero state. A ***programming operation*** changes the state of a cell from zero to one. This operation is irreversible in the sense that one cannot change the cell state from one to zero unless the entire memory is first erased. The ***memory-state vectors*** are all the binary vectors of length n, $\{0,1\}^n$. The ***data vectors*** are the set of all binary vectors of length k, $\{0,1\}^k$. Any WOM-code $\mathcal{C}$ is specified by its encoding map $\mathcal{E}_\mathcal{C}$ and decoding map $\mathcal{D}_\mathcal{C}$. The ***decoding map*** $\mathcal{D}_\mathcal{C} : \{0,1\}^n \to \{0,1\}^k$ assigns to each memory-state vector $\boldsymbol{c} \in \{0,1\}^n$ its corresponding data vector $\boldsymbol{v} = \mathcal{D}_\mathcal{C}(\boldsymbol{c}) \in \{0,1\}^k$. The ***encoding map*** $\mathcal{E}_\mathcal{C} : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n \cup \{\mathsf{E}\}$ indicates for each new data vector $\boldsymbol{v} \in \{0,1\}^k$ and memory-state vector $\boldsymbol{c} \in \{0,1\}^n$, a new memory-state vector $\boldsymbol{c}' = \mathcal{E}_\mathcal{C}(\boldsymbol{v},\boldsymbol{c})$ such that $\mathcal{D}_\mathcal{C}(\boldsymbol{c}') = \boldsymbol{v}$, and $c_i \leqslant c_i'$, for all $1 \leqslant i \leqslant n$. In case such a $\boldsymbol{c}' \in \{0,1\}^n$ does not exist, the value of the encoding map is $\mathcal{E}_\mathcal{C}(\boldsymbol{v},\boldsymbol{c}) = \mathsf{E}$.

Definition 4.2.1. *An $[n,k,t]$* ***WOM-code*** *$\mathcal{C}(\mathcal{E}_\mathcal{C},\mathcal{D}_\mathcal{C})$ is a coding scheme which consists of n cells and is defined by its encoding and decoding maps, denoted by $\mathcal{E}_\mathcal{C}$ and $\mathcal{D}_\mathcal{C}$, respectively. The WOM-code $\mathcal{C}$ guarantees any t writes of a k-bit data vector $\boldsymbol{v}$ without producing the block erasure symbol E. The* ***rate*** *of the WOM-code $\mathcal{C}$ is defined as $\mathcal{R} = \frac{kt}{n}$.*

Remark 4.2.1. As described in Chapter 3, it is possible to generalize the definition of WOM-codes to allow an arbitrary number of bits or symbols to be stored at each write. In this work we focus only on the case where the *same number of bits* is written at each write, i.e. fixed-rate WOM-codes. Since the rates on all individual writes are the same the “sum-rate” in the general case is simply called the “rate”. However, we note that it is possible to change the constructions to support the case where a different number of bits is written on each write.

The following definitions are also used in our work:

1. An $[n,k,t]$ WOM-code that can correct e errors is called an $[n,k,t]$ ***e-error-correcting WOM-code***.

2. An $[n,k,t]$ WOM-code that can detect e errors is called an $[n,k,t]$ ***e-error-detecting WOM-code***.

The definition of the decoding map in the second case is extended to be $\mathcal{D}_{\mathcal{C}_W} : \{0,1\}^n \to \{0,1\}^k \cup \{\mathtt{F}\}$, where the symbol $\mathtt{F}$ indicates an error detection flag.

Remark 4.2.2. If after decoding on the i-th write, a cell which is in state zero is erroneous, this error can be corrected (at least theoretically) prior to the next write by changing the state of this cell to a one. However, if after decoding on the i-th write, a cell which is in state one is erroneous, the state of this cell cannot be changed prior to the next write. In this case, however, it is assumed that on the $(i+1)$-st write the encoder knows that the cell's true state is a zero. There is no problem if the encoder wants to write a one in this cell. However, if the encoder wants to write a zero in this cell, then the error which was corrected on the i-th write will also occur on the $(i+1)$-st write because in this case it is not possible to physically change the cell's state . When we say that a WOM-code is an e-error-correcting code we mean that the code will correct e or a fewer errors on each write but we realize that some the errors which were corrected on one write could appear on subsequent writes. This information could be used in decoding but the decoder we consider here does not do so. We also assume here that there are no reading errors, that is, the correct state of a cell is always read.

In this work, we present error-detecting and error-correcting WOM-codes, which have the following generic structure:

1. We assume that there exists an $[n, k, t]$ WOM-code $\mathcal{C}(\mathcal{E}_{\mathcal{C}}, \mathcal{D}_{\mathcal{C}})$. Its n cells are denoted by $\boldsymbol{c} = (c_0, \ldots, c_{n-1})$ and called the ***information cells***. Note that this original code $\mathcal{C}$ *cannot* correct errors.

2. The constructed code consists of the n information cells $\boldsymbol{c}$, and r additional cells, called the ***redundancy cells***, and denoted by $\boldsymbol{p} = (p_0, \ldots, p_{r-1})$. The redundancy cells enable the decoder to correct cell-errors. That is, we get an $[n+r, k, t]$ WOM-code with some error correction/detection capabilities.

Two constructions of error-correcting WOM-codes were given in [20]. Both constructions correct a single cell-error during the writes. The first construction, based upon a double-error-correcting BCH code, enables one to write k bits using $n = 2^k - 1$ cells $t \approx n/15.42$ times, so its rate is roughly $\frac{k}{15.42} \approx \frac{\log_2(15.42t+1)}{15.42}$. The second construction, which uses the same number of cells, is based on a triple-error-correcting BCH code and stores $2k$ bits $t \approx n/26.9$ times. Its rate is approximately $\frac{2k}{26.9} \approx \frac{\log_2(26.9t+1)}{13.45}$. As in Chapter 3, in order to analyze the performance of WOM-codes, we find that the appropriate figure of merit is to compare the rates under the assumption of a fixed number of writes. In general, the more writes the WOM-code can support, the better the rate it can achieve. The second construction in [20] is superior to the first one as it achieves a better rate even though its number of writes is smaller.

A simple scheme to construct an e-error-correcting WOM-code is based upon an existing WOM-code $\mathcal{C}$ that stores k bits t times in n cells. In this scheme, each one of the n cells is replicated $2e+1$ times so it is possible to correct any e or fewer cell-errors. If the WOM-code $\mathcal{C}$ has rate $\mathcal{R} = \frac{kt}{n}$, then the generated e-error-correcting WOM-code has rate $\frac{1}{2e+1}\mathcal{R} = \frac{kt}{(2e+1)n}$. For example, in [5], a WOM-code which stores k bits $t = 5 \cdot 2^{k-4} + 1$ times using $n = 2^k - 1$ cells, for $k \geqslant 4$ is presented. If we use this WOM-code to construct a single-error-correcting WOM-code, then its rate, $\frac{1}{3}\frac{k(5\cdot 2^{k-4}+1)}{2^k-1} > \frac{\log_2(3.2(t-1))}{9.6}$, outperforms for t large enough the rate of the two other constructions in [20].

4.3 Single-Error-Detecting WOM-Codes

In this section we present single-error-detecting WOM-codes. As described in Section 5.2, we let $\mathcal{C}_{\mathcal{W}}(\mathcal{E}_{\mathcal{C}_{\mathcal{W}}}, \mathcal{D}_{\mathcal{C}_{\mathcal{W}}})$ be an $[n,k,t]$ WOM-code, and its cells, called the information cells, are denoted by $\boldsymbol{c} = (c_0, \ldots, c_{n-1})$. We construct an $[n+t,k,t]$ single-error-detecting WOM-code, denoted by $\mathcal{C}_{\text{SED}}(\mathcal{E}_{\mathcal{C}_{\text{SED}}}, \mathcal{D}_{\mathcal{C}_{\text{SED}}})$.

In this construction there are t redundancy cells, denoted by $\boldsymbol{p} = (p_0, p_1, \ldots, p_{t-1})$, i.e., the value of r in the general structure is t. The code $\mathcal{C}_{\text{SED}}$ satisfies the following property: at each write, the parity of the t redundancy cells, $\sum_{i=0}^{t-1} p_i$, and the parity of the n information cells, $\sum_{i=0}^{n-1} c_i$, are the same.

Theorem 4.3.1. *If $\mathcal{C}_{\mathcal{W}}$ is an $[n,k,t]$ WOM-code, then $\mathcal{C}_{SED}$ is an $[n+t,k,t]$ single-error-detecting WOM-code.*

Proof. We prove this theorem by showing the correctness of the encoding and decoding maps. In the encoding map $\mathcal{E}_{\mathcal{C}_{\text{SED}}}$, the new data vector $\boldsymbol{v}$ is encoded in the n information cells by the encoding map $\mathcal{E}_{\mathcal{C}_{\mathcal{W}}}(\boldsymbol{c}, \boldsymbol{v})$. If the parity of the information cells is changed, then one of the t redundancy cells is programmed. Since there are initially t redundancy cells in state zero and each time at most one of them is programmed, there is at least one unprogrammed cell at each write.

In the decoding map $\mathcal{D}_{\mathcal{C}_{\text{SED}}}$, at most one of the cells is in error. If the information cell's parity is different than the redundancy cell's parity, then the flag $\mathbf{F}$ is returned to indicate a single error detection. Otherwise, the data vector $\boldsymbol{v}$ is simply decoded by the decoding map $\boldsymbol{v} = \mathcal{D}_{\mathcal{C}_{\mathcal{W}}}(\boldsymbol{c})$. $\blacksquare$

This scheme can be applied to all known WOM-codes. In particular, the next example shows how to adapt the scheme to WOM-codes which are based on Hamming codes [2,5].

Example 4.3.1. In [2], a construction of WOM-codes, based on Hamming codes, is presented. For $k \geqslant 4$, the construction gives a $[2^k - 1, k, 2^{k-2} + 2]$ WOM-code, and for $k = 2, 3$ a $[2^k - 1, k, 2^{k-2} + 1]$ WOM-code. In particular, the $[3, 2, 2]$ WOM-code, presented by Rivest and Shamir [5], is a special case of this construction for $k = 2$. Later, in [5] the case $k \geqslant 4$ was improved and $[2^k - 1, k, 5 \cdot 2^{k-4} + 1]$ WOM-codes were presented.

For $k \geqslant 4$, Zémor showed that it is possible to change the construction such that, excluding the first write, the number of programmed cells at each write is even [21]. Therefore, the parity bit changes its values at most once. Thus, one redundancy cell is sufficient for the construction and we get a $[2^k, k, 5 \cdot 2^{k-4} + 1]$ single-error-detecting code. In fact, a similar construction to this code with the same parameters was presented by Zémor in [21].

For $k = 2, 3$, the construction is slightly modified. At each write, the redundancy cells' parity is the complement of the information cells' parity. Then, at most $2^{k-2} = t - 1$ cells are sufficient and thus a $[2^k + 2^{k-2} - 1, k, 2^{k-2} + 1]$ single-error-detecting code exists. The following table demonstrates the construction for the $[4, 2, 2]$ single-error-detecting WOM-code. The bold font represents the bit in the redundancy cell. A similar table can be built for the $[9, 3, 3]$ single-error-detecting WOM-code.

Bits Value	First Write	Second Write
00	000**1**	111**0**
01	001**0**	110**1**
10	010**0**	101**1**
11	100**0**	011**1**

4.4 Single-Error-Correcting WOM-Codes

In order to construct single-error-correcting WOM-codes, we start as in Section 4.3 with an $[n, k, t]$ WOM-code, $\mathcal{C}_{\mathcal{W}}(\mathcal{E}_{\mathcal{C}_{\mathcal{W}}}, \mathcal{D}_{\mathcal{C}_{\mathcal{W}}})$. Its information cells are $\boldsymbol{c} = (c_0, \ldots, c_{n-1})$ and we add r redundancy cells, $\boldsymbol{p} = (p_0, \ldots, p_{r-1})$, that form a word in $\mathcal{C}_{\mathcal{WD}}(\mathcal{E}_{\mathcal{C}_{\mathcal{WD}}}, \mathcal{D}_{\mathcal{C}_{\mathcal{WD}}})$, an $[r, \lceil \log_2(n+1) \rceil, t]$ single-error-detecting WOM-code. Then, we construct an $[n + r, k, t]$ single-error-correcting WOM-code, denoted by $\mathcal{C}_{\text{SEC}}(\mathcal{E}_{\mathcal{C}_{\text{SEC}}}, \mathcal{D}_{\mathcal{C}_{\text{SEC}}})$, as follows.

At each write we generate a $\lceil \log_2(n+1) \rceil$-bit vector, called the ***syndrome*** and denoted by $\boldsymbol{s}$. The syndrome will correspond to the redundancy bits of a Hamming code (or a shortened Hamming code) of length n, and will make it possible to locate an information cell in error.

Next, and in the following sections, we provide the exact specification of the given error-correcting WOM-codes by their encoding and decoding maps. These maps are described algorithmically using a pseudo-code notation. In this specification we will use the encoding and decoding maps $\mathcal{E}_{\mathcal{C}_\mathcal{W}}, \mathcal{D}_{\mathcal{C}_\mathcal{W}}$ of the WOM-code $\mathcal{C}_\mathcal{W}$ and the encoding and decoding maps $\mathcal{E}_{\mathcal{C}_{\mathcal{WD}}}, \mathcal{D}_{\mathcal{C}_{\mathcal{WD}}}$ of the single-error-detecting WOM-code $\mathcal{C}_{\mathcal{WD}}$. We let α be a primitive element in the extension field $GF(2^{\lceil \log_2(n+1) \rceil})$.

Encoding map $\mathcal{E}_{\mathcal{C}_{\text{SEC}}}$**:** The input is the memory-state vector $(\boldsymbol{c}, \boldsymbol{p})$ and the new k-bit data vector $\boldsymbol{v}$. The output is either a new memory-state vector $(\boldsymbol{c}', \boldsymbol{p}')$ or the erasure symbol E.

```
1. $c' = \mathcal{E}_{\mathcal{C}_\mathcal{W}}(c, v)$;
2. if ($c'$ == E) return E;
3. $s = \sum_{i=0}^{n-1} c'_i \alpha^i$;
4. $p' = \mathcal{E}_{\mathcal{C}_{\mathcal{WD}}}(p, s)$;
5. if ($p'$ == E) return E;
6. return ($c', p'$);
```

In the encoding map, $\mathcal{E}_{\mathcal{C}_{\text{SEC}}}$, the data vector $\boldsymbol{v}$ is encoded in the information cells $\boldsymbol{c}$ (line 1). If writing does not succeed, the symbol E is returned (line 2). Otherwise, the syndrome $\boldsymbol{s}$ of the new n information cells is calculated (line 3). Then, $\boldsymbol{s}$ is encoded in the redundancy cells using the encoding map $\mathcal{E}_{\mathcal{C}_{\mathcal{WD}}}(\boldsymbol{p}, \boldsymbol{s})$ (line 4). If this writing fails, the symbol E is returned (line 5); otherwise, the new memory-state vector is returned (line 6). Note that since the encoding map $\mathcal{E}_{\mathcal{C}_\mathcal{W}}$ can write t messages of k-bits each and the encoding map $\mathcal{E}_{\mathcal{C}_{\mathcal{WD}}}$ can write t times the $\lceil \log_2(n+1) \rceil$-bit syndrome $\boldsymbol{s}$, the encoding map $\mathcal{E}_{\mathcal{C}_{\text{SEC}}}$ also can write k-bits t times.

Decoding map $\mathcal{D}_{\mathcal{C}_{\text{SEC}}}$**:** The input is the memory-state vector $(\boldsymbol{c}', \boldsymbol{p}')$. The output is the decoded k-bit data vector $\boldsymbol{v}$.

```
1. $s'' = \mathcal{D}_{\mathcal{C}_{\mathcal{WD}}}(p')$;
2. if ($s''$ == F)
3.    { $v = \mathcal{D}_{\mathcal{C}_\mathcal{W}}(c')$; return $v$; }
4. $s' = \sum_{i=0}^{n-1} c'_i \alpha^i$;
5. if ($s' == s''$)
6.    { $v = \mathcal{D}_{\mathcal{C}_\mathcal{W}}(c')$; return $v$; }
7. $i = \log_\alpha(s' + s'')$;
8. $v = \mathcal{D}_{\mathcal{C}_\mathcal{W}}(c'_0, \ldots, c'_{i-1}, 1 - c'_i, c'_{i+1}, \ldots, c'_{n-1})$;
9. return $v$;
```

The syndrome $\boldsymbol{s}''$ is decoded by applying the decoding map $\mathcal{D}_{\mathcal{C}_{\mathcal{WD}}}$ on the redundancy cells $\boldsymbol{p}'$ (line 1). The code $\mathcal{C}_{\mathcal{WD}}$ is a single-error-detecting WOM-code and hence by its decoding map $\mathcal{D}_{\mathcal{C}_{\mathcal{WD}}}$ it is possible to determine if there is an error in one of the r redundancy cells (line 2). We distinguish between the following two cases:

1. If one of the redundancy cells is in error, i.e. the condition in line 2 holds, then there is no error in the information cells and $\boldsymbol{v}$ is decoded by the decoding map $\mathcal{D}_{\mathcal{C}_{\mathcal{W}}}$ (line 3).
2. If there is no error in the redundancy cells, then $\boldsymbol{s}''$ is the correct value of the syndrome $\boldsymbol{s}$. The received syndrome $\boldsymbol{s}'$ from the received n information cells is $\boldsymbol{s}' = \sum_{i=0}^{n-1} c'_i \alpha^i$ (line 4). If $\boldsymbol{s}' = \boldsymbol{s}''$ (line 5), then there is no error in the n information cells and it is possible to decode the correct value of the data vector $\boldsymbol{v}$ (line 6). Otherwise, if the i-th cell is in error, then $\boldsymbol{s}' + \boldsymbol{s}'' = \alpha^i$. The calculation of $\log_\alpha(\boldsymbol{s}' + \boldsymbol{s}'')$ returns the value i such that $\alpha^i = \boldsymbol{s}' + \boldsymbol{s}''$ (line 7). This identifies the erroneous cell and again can decode the data vector $\boldsymbol{v}$ (line 8).

Thus we have proved the following theorem.

Theorem 4.4.1. *If $\mathcal{C}_{\mathcal{W}}$ is an $[n,k,t]$ WOM-code, $\mathcal{C}_{\mathcal{WD}}$ is an $[r, \lceil \log_2(n+1) \rceil, t]$ single-error-detecting WOM-code, then $\mathcal{C}_{SEC}$ is an $[n+r,k,t]$ single-error-correcting WOM-code.*

The next example demonstrates how to use this construction to build specific single-error-correcting WOM-codes.

Example 4.4.1. As in Example 4.3.1, the code $\mathcal{C}_{\mathcal{W}}$ is chosen to be the $[2^k-1, k, 5\cdot 2^{k-4}+1]$ WOM-code for $k \geqslant 4$ from [5]. Therefore, $n = 2^k - 1$, and $\lceil \log_2(n+1) \rceil = k$, so we can use the $[2^k, k, 5\cdot 2^{k-4}+1]$ single-error-detecting WOM-code from Example 4.3.1. The resulting $[2\cdot 2^k - 1, k, 5\cdot 2^{k-4}+1]$ single-error-correcting WOM-code has rate

$$\mathcal{R} = \frac{k(5\cdot 2^{k-4}+1)}{2\cdot 2^k - 1} > \frac{\log_2(3.2(t-1))}{6.4},$$

which is an improvement upon the constructions in [20] and the simple construction presented in the Introduction.

4.5 Double-Error-Correcting WOM-Codes

The double-error-correcting WOM-codes construction is very similar to the single-error-correcting case in Section 4.4, where the same WOM-codes $\mathcal{C}_{\mathcal{W}}, \mathcal{C}_{\mathcal{WD}}$ are used. There are $2r$ redundancy cells, partitioned into two r-cell groups, $\boldsymbol{p}_1 = (p_0, p_1, \ldots, p_{r-1})$ and $\boldsymbol{p}_2 =$

$(p_r, p_1, \ldots, p_{2r-1})$. The redundancy groups $\boldsymbol{p}_1$ and $\boldsymbol{p}_2$ store $\lceil \log_2(n+1) \rceil$-bit syndrome vectors $\boldsymbol{s}_1$ and $\boldsymbol{s}_2$, respectively. The two syndromes correspond to the two roots α, α^3 of a double-error-correcting BCH code, denoted by $\mathcal{C}_{\text{2-BCH}}$, where α is a primitive element in the field $GF(2^{\lceil \log_2(n+1) \rceil})$. In this construction, $\lceil \log_2(n+1) \rceil$ is assumed to be an odd integer. The code is denoted by $\mathcal{C}_{\text{DEC}}(\mathcal{E}_{\mathcal{C}_{\text{DEC}}}, \mathcal{D}_{\mathcal{C}_{\text{DEC}}})$.

Encoding map $\mathcal{E}_{\mathcal{C}_{\text{DEC}}}$**:** The input is the memory-state vector $(\boldsymbol{c}, \boldsymbol{p}_1, \boldsymbol{p}_2)$ and the new k-bit data vector $\boldsymbol{v}$. The output is either a new memory-state vector $(\boldsymbol{c}', \boldsymbol{p}_1', \boldsymbol{p}_2')$ or the erasure symbol E.

```
1. $c' = \mathcal{E}_{\mathcal{C}_{\mathcal{W}}}(c, v)$;
2. if ($c'$ == E) return E;
3. $s_1 = \sum_{i=0}^{n-1} c_i' \alpha^i$; $s_2 = \sum_{i=0}^{n-1} c_i' \alpha^{3i}$;
4. $p_1' = \mathcal{E}_{\mathcal{C}_{\mathcal{WD}}}(p_1, s_1)$; $p_2' = \mathcal{E}_{\mathcal{C}_{\mathcal{WD}}}(p_2, s_2)$;
5. if (($p_1'$ == E)OR($p_2'$ == E)) return E;
6. return ($c', p_1', p_2'$);
```

The new k-bit data vector $\boldsymbol{v}$ is encoded in the information cells using the encoding map $\mathcal{E}_{\mathcal{C}_{\mathcal{W}}}$ (line 1). The success of this writing is then checked (line 2). The two syndromes $\boldsymbol{s}_1, \boldsymbol{s}_2$ are calculated (line 3) and are encoded in the redundancy cells (line 4) while checking the writing success (line 5). If the last two writing operations succeed, the encoding map returns the new memory-state vector (line 6).

For the decoding map $\mathcal{D}_{\mathcal{C}_{\text{DEC}}}$, we use the single-error-correcting WOM-code decoding map $\mathcal{D}_{\mathcal{C}_{\text{SEC}}}$, which receives as its input n information cells and r redundancy cells. Note that while the code $\mathcal{C}_{\text{SEC}}$ uses a fixed primitive element $\alpha \in GF(2^{\lceil \log_2(n+1) \rceil})$, it is possible to use any other primitive element in the field $GF(2^{\lceil \log_2(n+1) \rceil})$. We slightly modify the input arguments of the decoding map $\mathcal{D}_{\mathcal{C}_{\text{SEC}}}$ such that the primitive element is its first parameter. The modified decoding map is denoted by $\mathcal{D}'_{\mathcal{C}_{\text{SEC}}}$. We use the decoding map $\mathcal{D}_{\mathcal{C}_{\text{2-BCH}}}$ of the double-error-correcting BCH code. Its input is the $2\lceil \log_2(n+1) \rceil$ syndrome bits; its output is the error vector.

Decoding map $\mathcal{D}_{\mathcal{C}_{\text{DEC}}}$**:** The input is the memory-state vector $(\boldsymbol{c}', \boldsymbol{p}_1', \boldsymbol{p}_2')$. The output is the decoded k-bit data vector $\boldsymbol{v}$.

```
1. $s''_1$ = $\mathcal{D}_{\mathcal{C}_{\mathcal{WD}}}$($p'_1$); $s''_2$ = $\mathcal{D}_{\mathcal{C}_{\mathcal{WD}}}$($p'_2$);
2. if ($s''_1$ == F)
3.   { $v$ = $\mathcal{D}'_{\mathcal{C}_{\text{SEC}}}$($\alpha^3, c', p'_2$); return $v$; }
4. if ($s''_2$ == F)
5.   { $v$ = $\mathcal{D}'_{\mathcal{C}_{\text{SEC}}}$($\alpha, c', p'_1$); return $v$; }
6. $s'_1$ = $\sum_{i=0}^{n-1} c'_i\alpha^i$; $s'_2$ = $\sum_{i=0}^{n-1} c'_i\alpha^{3i}$;
7. if (($s'_1$ == $s''_1$) OR ($s'_2$ == $s''_2$))
8.   { $v$ = $\mathcal{D}_{\mathcal{C}_{\mathcal{W}}}$($c'$); return $v$; }
9. $e'$ = $\mathcal{D}_{\mathcal{C}_{\text{2-BCH}}}$($s'_1 + s''_1, s'_2 + s''_2$);
10. $v$ = $\mathcal{D}_{\mathcal{C}_{\mathcal{W}}}$($c' + e'$);
11. return $v$;
```

The two syndromes s''_1, s''_2 are decoded using the redundancy cells and the decoding map $\mathcal{D}_{\mathcal{C}_{\mathcal{WD}}}$ (line 1). If $s''_1 = \mathbf{F}$ (line 2) then there is at least one error in the redundancy cells of group p'_1, and at most one error in the information cells c' and the second redundancy group p'_2. Therefore, it possible to decode the data vector v by applying the decoding map $\mathcal{D}'_{\mathcal{C}_{\text{SEC}}}$ to the cells in c' and p'_2 while taking α^3 to be the primitive element (line 3). Note that since $\lceil \log_2(n+1) \rceil$ is an odd integer, α^3 is also a primitive element in $GF(2^{\lceil \log_2(n+1) \rceil})$. Similarly, if $s''_2 = \mathbf{F}$ (line 4), then we decode by applying the decoding map $\mathcal{D}'_{\mathcal{C}_{\text{SEC}}}$ to the cells c' and p'_1, while α is the primitive element (line 5).

If according to the decoding map $\mathcal{D}_{\mathcal{C}_{\mathcal{WD}}}$, no error is decoded in both the redundancy cell groups, then either there is no error in all the redundancy cells or there are exactly two errors in one of the two redundancy cell groups. First, the syndromes s'_1, s'_2 from the received n information cells are calculated (line 6). Then, we consider the following two cases:

1. If $s'_1 = s''_1$ or $s'_2 = s''_2$ (line 7), then necessarily there is no error in the n information cells and the k-bit data vector is calculated and returned (line 8). (This is true since if there is at least one error in the information cells then there is no error in the redundancy cells and neither of these equalities holds, which is a contradiction.)

2. If $s'_1 \neq s''_1$ and $s'_2 \neq s''_2$ (line 9) then at least one error occurred in the n information cells and no errors in the redundancy cells. The error vector is found by applying the decoding algorithm of the two-error-correcting BCH code, $\mathcal{D}_{\mathcal{C}_{\text{2-BCH}}}$, to $s'_1 + s''_1$ and $s'_2 + s''_2$ (line 9). Then, we know the correct value of the n information cells and it is again possible to successfully decode the data vector v (line 10).

We conclude this construction in the following theorem.

Theorem 4.5.1. *If $\mathcal{C}_{\mathcal{W}}$ is an $[n,k,t]$ WOM-code, $\mathcal{C}_{\mathcal{WD}}$ is an $[r, \lceil \log_2(n+1) \rceil, t]$ single-error-detecting WOM-code, and $\lceil \log_2(n+1) \rceil$ is an odd integer, then $\mathcal{C}_{DEC}$ is an $[n+2r,k,t]$ double-error-correcting WOM-code.*

The construction does not work if $\lceil \log_2(n+1) \rceil$ is an even integer since α^3 is no longer a primitive element in the field $GF(2^{\lceil \log_2(n+1) \rceil})$, and thus the decoding map in line 3 cannot succeed. Clearly, it is possible to modify it by working over the field $GF(2^{1+\lceil \log_2(n+1) \rceil})$ and storing syndromes of $1+\lceil \log_2(n+1) \rceil$ bits. Next, we show a modification in case that $\lceil \log_2(n+2) \rceil$ is an even integer by adding t more cells.

Assume that $\lceil \log_2(n+2) \rceil$ is an even integer. The last construction is modified and we present its differences in the encoding and decoding maps. The main modifications are as follows:

1. Instead of using the $[n,k,t]$ WOM-code $\mathcal{C}_{\mathcal{W}}$, an $[n+t,k,t]$ single-error-detecting WOM-code is used and we denote it by $\mathcal{C}'_{\mathcal{W}}(\mathcal{E}_{\mathcal{C}'_{\mathcal{W}}}, \mathcal{D}_{\mathcal{C}'_{\mathcal{W}}})$. The t additional redundancy cells are denoted by $\boldsymbol{q} = (q_0, \dots, q_{t-1})$.

2. Instead of using the root α^3 we use the root α^{-1}.

3. The syndromes $\boldsymbol{s}_1$ and $\boldsymbol{s}_2$ are calculated according to the new roots applied to the information cells $\boldsymbol{c}$ and their parity value, which is stored in the new redundancy cells $\boldsymbol{q}$.

The input and output to the encoding map are changed accordingly where the memory-state vector is $(\boldsymbol{c}, \boldsymbol{q}, \boldsymbol{p}_1, \boldsymbol{p}_2)$. In the first and second lines, we use the encoding map $\mathcal{E}_{\mathcal{C}'_{\mathcal{W}}}$ instead of $\mathcal{E}_{\mathcal{C}_{\mathcal{W}}}$ on the cells $(\boldsymbol{c}, \boldsymbol{q})$. The syndrome values in line 3 and the returned new memory-state vector in line 6 are also changed accordingly.

```
1. $(c',q') = \mathcal{E}_{\mathcal{C}'_{\mathcal{W}}}((c,q),v)$;
2. if ($(c',q')$ == E) return E;
3. $s_1 = \sum_{i=0}^{n-1} c'_i\alpha^i + (\sum_{i=0}^{t-1} q'_i)\alpha^n$;
   $s_2 = \sum_{i=0}^{n-1} c'_i\alpha^{-i} + (\sum_{i=0}^{t-1} q'_i)\alpha^{-n}$;
6. return $(c',q',p'_1,p'_2)$;
```

The decoding algorithm is also very similar. Since we use the root α^{-1} and also the value of the t new redundancy cells, lines 3 and 6 are changed as follows. Note that α^{-1} is also a primitive element and therefore the decoding map in line 3 succeeds.

```
3.   { $v$ = $\mathcal{D}'_{\mathcal{C}_{\text{SEC}}}$($\alpha^{-1}$,$c'$,$p'_2$); return $v$; }
6.  $s'_1$ = $\sum_{i=0}^{n-1} c'_i\alpha^i$ + ($\sum_{i=0}^{t-1} q'_i$)$\alpha^n$;
    $s'_2$ = $\sum_{i=0}^{n-1} c'_i\alpha^{-i}$ + ($\sum_{i=0}^{t-1} q'_i$)$\alpha^{-n}$;
```

If the decoder reaches line 9, then there is at least one error in the n information cells $\boldsymbol{c}'$ and t redundancy cells $\boldsymbol{q}'$. The main difference in the decoding is that at this line we necessarily need to know if there is one or two cells in error among the n information cells $\boldsymbol{c}'$ and t redundancy cells $\boldsymbol{q}'$. If there is a single error, that is, the parity of the n information cells and the parity of the t additional redundancy cells are not the same (line 9), then we can decode the data vector using the decoding map $\mathcal{D}'_{\mathcal{C}_{\text{SEC}}}$ with the root α since there is at most one error in the information cells and no error in the redundancy cells $\boldsymbol{p}'_1$ (line 10). Otherwise, there are exactly two errors in the n information cells and t redundancy cells. The values of $\boldsymbol{e}_1$ and $\boldsymbol{e}_2$ which are calculated in line 11 are of the form

$$\boldsymbol{e}_1 = \alpha^i + \alpha^j,\ \boldsymbol{e}_2 = \alpha^{-i} + \alpha^{-j},$$

for some $0 \leqslant i, j \leqslant n, i \neq j$, and

$$\begin{aligned}
&\boldsymbol{e}_1(\boldsymbol{e}_1^2 + \boldsymbol{e}_1\boldsymbol{e}_2^{-1})\\
&= (\alpha^i + \alpha^j)\left((\alpha^i + \alpha^j)^2 + (\alpha^i + \alpha^j)\frac{\alpha^{i+j}}{\alpha^i + \alpha^j}\right)\\
&= (\alpha^i + \alpha^j)\left(\alpha^{2i} + \alpha^{2j} + \alpha^{i+j}\right) = \alpha^{3i} + \alpha^{3j}.
\end{aligned}$$

Therefore, the values of i and j, i.e. the error vector, can be found by applying the decoding procedure $\mathcal{D}_{\mathcal{C}_{\text{2-BCH}}}$ to $\boldsymbol{e}_1$ and $\boldsymbol{e}_1(\boldsymbol{e}_1^2 + \boldsymbol{e}_1\boldsymbol{e}_2^{-1})$ (line 12). Next, the data vector can be successfully decoded (line 13). Note that the error vector in line 12 consists of $n + 1$ bits while for the decoding map in line 13 we need only its first n bits.

```
9.  if (($\sum_{i=0}^{n-1} c'_i$) != ($\sum_{i=0}^{t-1} q'_i$))
10.   { $v$ = $\mathcal{D}'_{\mathcal{C}_{\text{SEC}}}$($\alpha$,$c'$,$p'_1$); return $v$; }
11.  $e_1$ = $s'_1$ + $s''_1$; $e_2$ = $s'_2$ + $s''_2$;
12.  $e'$ = $\mathcal{D}_{\mathcal{C}_{\text{2-BCH}}}$($e_1$,$e_1$($e_1^2$ + $e_1e_2^{-1}$));
13.  $v$ = $\mathcal{D}_{\mathcal{C}_{\mathcal{W}}}$($c'$ + $e'$);
14.  return $v$;
```

To conclude, we state the following theorem.

Theorem 4.5.2. *Let $\mathcal{C}_{\mathcal{W}}$ be an $[n,k,t]$ WOM-code and $\mathcal{C}_{\mathcal{WD}}$ be an $[r, \lceil \log_2(n+2) \rceil, t]$ single-error-detecting WOM-code. Suppose $\lceil \log_2(n+2) \rceil$ is an even integer. Then there exists an $[n+2r+t,k,t]$ double-error-correcting WOM-code.*

4.6 Triple-Error Correcting WOM-Codes

From the previous sections we might think that a general scheme to construct an e-error correcting WOM-code is to combine an existing WOM-code and a cyclic e-error-correcting code, where the latter code is defined by e roots $\alpha_1,\dots,\alpha_e$. However, not every e-error-correcting code would work this scheme. For example, in the double-error-correcting construction in Section 4.5, the BCH code with roots α and α^3 cannot work if $\lceil \log_2(n+1) \rceil$ is an even integer. This results from the fact that α^3 is not a primitive element and hence the code generated only by α^3 is not a single-error-correcting code. For arbitrary e, if the cyclic e-error-correcting code is defined by e roots, then a necessary but not sufficient condition for this scheme to work is that every subset of $k \leqslant e$ roots generates a cyclic k-error-correcting code. We state this property in the following definition.

Definition 4.6.1. *Let n be an integer and $\alpha_1,\dots,\alpha_e$ be e different elements in the field $GF(2^n)$. Let the code $\mathcal{C}(\alpha_1,\dots,\alpha_e)$ be a cyclic error-correcting code of length 2^n-1, which its roots are $\alpha_1,\dots,\alpha_e$. The code $\mathcal{C}(\alpha_1,\dots,\alpha_e)$ is called a **strong e-error-correcting code** if for every $1 \leqslant k \leqslant e$ and every set of k distinct elements $\alpha_{i_1},\dots,\alpha_{i_k} \in \{\alpha_1,\dots,\alpha_e\}$, the code $\mathcal{C}(\alpha_{i_1},\dots,\alpha_{i_k})$ is a k-error-correcting code.*

We note that finding strong e-error-correcting codes is a fascinating problem by itself but is beyond the scope of this work. Next, we show how to choose the roots $\alpha_1,\alpha_2,\alpha_3$ such that $\mathcal{C}(\alpha_1,\alpha_2,\alpha_3)$ is a strong triple-error-correcting code. For the following discussion, α is assumed to be a primitive element in $GF(2^n)$. The following result was proved by Kasami in [10].

Theorem 4.6.2. *[10]. Let n be an odd integer and $\gcd(n,k)=1$. Then, $\mathcal{C}(\alpha,\alpha^{2^k+1},\alpha^{2^{3k}+1})$ is a cyclic triple-error-correcting code.*

In [1], the authors show an alternative proof to the last theorem and state the following lemma.

Lemma 4.6.3. *Let n be an integer and $\gcd(n,k)=1$. Then, $\mathcal{C}(\alpha,\alpha^{2^k+1})$ is a cyclic double-error-correcting code.*

These two results imply the following lemma.

Lemma 4.6.4. *Let n be an integer such that $\gcd(n,6)=1$, and let $k=\frac{n-1}{2}$. Then, the following properties hold.*

1. *The codes $\mathcal{C}(\alpha),\mathcal{C}(\alpha^{2^k+1}),\mathcal{C}(\alpha^{2^{3k}+1})$ are cyclic single-error-correcting codes.*

2. *The codes $\mathcal{C}(\alpha,\alpha^{2^k+1}),\mathcal{C}(\alpha,\alpha^{2^{3k}+1})$ are cyclic double-error-correcting codes.*

3. *The code $\mathcal{C}(\alpha,\alpha^{2^k+1},\alpha^{2^{3k}+1})$ is a cyclic triple-error-correcting code.*

Proof.

1. Since $k=\frac{n-1}{2}$, we know that 2^k+1 is a divisor of $2^{n-1}-1$. Since $\gcd(2^n-1,2^{n-1}-1)=1$, we conclude that $\gcd(2^n-1,2^k+1)=1$. Therefore α^{2^k+1} is a primitive element in $GF(2^n)$ and the code $\mathcal{C}(\alpha^{2^k+1})$ is a cyclic single-error-correcting code. Since $\gcd(n,6)=1$, it follows also that $\gcd(2^n-1,2^{3k}+1)=1$, and therefore the code $\mathcal{C}(\alpha^{2^{3k}+1})$ is a cyclic single-error-correcting code as well.

2. Since $\gcd(n,k)=1$, the condition of Lemma 4.6.3 holds and the code $\mathcal{C}(\alpha,\alpha^{2^k+1})$ is a double-error-correcting code. Similarly, since $\gcd(n,6)=1$ and $\gcd(n,k)=1$, it follows that $\gcd(n,3k)=1$, and again by Lemma 4.6.3, the code $\mathcal{C}(\alpha,\alpha^{2^{3k}+1})$ is a double-error-correcting code.

3. Since $\gcd(n,6)=1$, n is necessarily an odd integer and since $\gcd(n,k)=1$ the conditions of Theorem 4.6.2 hold. Therefore, the code $\mathcal{C}(\alpha,\alpha^{2^k+1},\alpha^{2^{3k}+1})$ is a triple-error-correcting code.

■

We note that at this point the code $\mathcal{C}(\alpha,\alpha^{2^k+1},\alpha^{2^{3k}+1})$ is “almost” a strong triple-error-correcting code. All that remains to be shown is that the code $\mathcal{C}(\alpha^{2^k+1},\alpha^{2^{3k}+1})$ is a double-error-correcting code. Before doing so, we state the definition of an almost perfect nonlinear mapping.

Definition 4.6.5. *A mapping $f: GF(p^n)\to GF(p^n)$ is called an **almost perfect nonlinear (APN)** mapping if each equation*

$$f(x+a)-f(x)=b$$

*for $a,b\in GF(p^n)$ and $a\neq 0$ has at most two solutions in $GF(p^n)$. If f is an APN mapping and is of the form $f(x)=x^d$ then f is called an **almost perfect nonlinear power mapping**.*

The next lemma was proved in [10].

Lemma 4.6.6. *If n is an odd integer, $2 \leqslant k \leqslant \frac{n-1}{2}$, and $\gcd(n,k) = 1$ then the mapping $f(x) = x^{2^{2k}-2^k+1}$ over $GF(2^n)$ is an APN mapping.*

The proof of the next lemma follows an outline similar to that of the proof of Theorem 1 in [1].

Lemma 4.6.7. *If n, k are integers, $\gcd(n,6) = 1$ and $k = \frac{n-1}{2}$, then $\mathcal{C}(\alpha^{2^k+1}, \alpha^{2^{3k}+1})$ is a double-error-correcting code.*

Proof. Note first that α^{2^k+1} is a primitive element in $GF(2^n)$ since $\gcd(2^n-1, 2^k+1) = 1$. Also, $\gcd(n,k) = 1$ and n is an odd integer, so, according to Lemma 4.6.6, $f(x) = x^d$ is an APN power mapping, where $d = 2^{2k} - 2^k + 1$. We denote $\gamma = \alpha^{2^k+1}$, and hence need to prove that $\mathcal{C}(\gamma, \gamma^d)$ is a double-error-correcting code.

Assume to the contrary that the code is not a double-error-correcting code. Clearly, there are no codewords of weight one or two and hence there exists a codeword of weight three or four. Assume there exist a codeword of weight four. Then, there exist four integers $0 \leqslant i_1 < i_2 < i_3 < i_4 \leqslant 2^n - 2$ such that

$$\gamma^{i_1} + \gamma^{i_2} + \gamma^{i_3} + \gamma^{i_4} = 0$$
$$(\gamma^{i_1})^d + (\gamma^{i_2})^d + (\gamma^{i_3})^d + (\gamma^{i_4})^d = 0.$$

The last two equations can be written as follows

$$\gamma^{i_1} + \gamma^{i_2} = a = \gamma^{i_3} + \gamma^{i_4}$$
$$(\gamma^{i_1})^d + (\gamma^{i_2})^d = b = (\gamma^{i_3})^d + (\gamma^{i_4})^d,$$

for some $a, b \in GF(2^n)$, and $a \neq 0$. Hence, the equation

$$(x+a)^d + x^d = b$$

has four different solutions: $\gamma^{i_1}, \gamma^{i_2}, \gamma^{i_3}, \gamma^{i_4}$. This is a contradiction since x^d is an APN mapping. The case of a codeword of weight three is handled similarly. ■

From Lemma 4.6.4 and Lemma 4.6.7 we conclude the following theorem.

Theorem 4.6.8. *If n, k are integers, $\gcd(n,6) = 1$, and $k = \frac{n-1}{2}$, then $\mathcal{C}(\alpha, \alpha^{2^k+1}, \alpha^{2^{3k}+1})$ is a strong triple-error-correcting code.*

We are now ready to show the triple-error-correcting WOM-code construction. Again, we use the WOM-codes $\mathcal{C}_{\mathcal{W}}, \mathcal{C}_{\mathcal{WD}}$, and assume that $\gcd(\lceil \log_2(n+1) \rceil, 6) = 1$ and α is a primitive element in $GF(2^{\lceil \log_2(n+1) \rceil})$. The strong triple-error-correcting code is denoted by $\mathcal{C}_3^{\text{strong}}(\mathcal{E}_{\mathcal{C}_3^{\text{strong}}}, \mathcal{D}_{\mathcal{C}_3^{\text{strong}}})$. Its roots are $\alpha_1 = \alpha, \alpha_2 = \alpha^{2^k+1}, \alpha_3 = \alpha^{2^{3k}+1}$, where $k = \frac{\lceil \log_2(n+1) \rceil - 1}{2}$. There are $3r + t$ redundancy cells, divided into four groups:

1. The first t cells $\boldsymbol{q} = (q_0, \ldots, q_{t-1})$ are used with the n information cells to construct an $[n+t, k, t]$ single-error-detecting WOM-code $\mathcal{C}'_W(\mathcal{E}_{\mathcal{C}'_W}, \mathcal{D}_{\mathcal{C}'_W})$.

2. The other three groups $\boldsymbol{p}_1 = (p_0, \ldots, p_{r-1})$, $\boldsymbol{p}_2 = (p_r, \ldots, p_{2r-1})$, and $\boldsymbol{p}_3 = (p_{2r}, \ldots, p_{3r-1})$ constitutes of r cells each. The i-th group, $i = 1, 2, 3$, stores the $\lceil \log_2(n+1) \rceil$-bit syndrome $\boldsymbol{s}_i$ which corresponds to the root α_i.

To conclude, we describe an $[n + t + 3r, k, t]$ triple-error-correcting WOM-code, which we denote by $\mathcal{C}_{\text{TEC}}(\mathcal{E}_{\mathcal{C}_{\text{TEC}}}, \mathcal{D}_{\mathcal{C}_{\text{TEC}}})$.

Encoding map $\mathcal{E}_{\mathcal{C}_{\text{TEC}}}$: The input is the memory-state vector $(\boldsymbol{c}, \boldsymbol{q}, \boldsymbol{p}_1, \boldsymbol{p}_2, \boldsymbol{p}_3)$ and the new k-bit data vector $\boldsymbol{v}$. The output is either a new memory-state vector $(\boldsymbol{c}', \boldsymbol{q}', \boldsymbol{p}'_1, \boldsymbol{p}'_2, \boldsymbol{p}'_3)$ or the erasure symbol E.

```
1. $(c',q')$ = $\mathcal{E}_{\mathcal{C}'_W}$(($c,q$),$v$);
2. if (($c',q'$) == E) return E;
3. $s_1$ = $\sum_{i=0}^{n-1} c'_i \alpha_1^{id_1}$; $s_2$ = $\sum_{i=0}^{n-1} c'_i \alpha_2^{id_2}$; $s_3$ = $\sum_{i=0}^{n-1} c'_i \alpha_3^{id_3}$;
4. $p'_1$ = $\mathcal{E}_{\mathcal{C}_{W\mathcal{D}}}$($p_1,s_1$); $p'_2$ = $\mathcal{E}_{\mathcal{C}_{W\mathcal{D}}}$($p_2,s_2$);
   $p'_3$ = $\mathcal{E}_{\mathcal{C}_{W\mathcal{D}}}$($p_3,s_3$);
5. if (($p'_1$ == E)OR($p'_2$ == E)OR($p'_3$ == E))
     return E;
6. return ($c',q',p'_1,p'_2,p'_3$);
```

The new k-bit data vector $\boldsymbol{v}$ is encoded in the information cells $\boldsymbol{c}$ and the first group of the redundancy cells $\boldsymbol{q}$ using the encoding map $\mathcal{E}_{\mathcal{C}'_W}$ (line 1). If this writing does not succeed the symbol E is returned (line 2). Otherwise, the three syndromes $\boldsymbol{s}_1, \boldsymbol{s}_2, \boldsymbol{s}_3$ are calculated from the n information cells (line 3) and are encoded in the last three groups of redundancy cells (line 4) while checking their writing success (line 5). If the last three writing operations succeed, the encoding map returns the new memory-state vector (line 6).

In the decoding map, $\mathcal{D}_{\mathcal{C}_{\text{TEC}}}$, we use the decoding map of the double-error-correcting WOM-code $\mathcal{D}_{\mathcal{C}_{\text{DEC}}}$. Note that in the decoding map $\mathcal{D}_{\mathcal{C}_{\text{DEC}}}$, instead of using a double-error-correcting BCH code, we can use any other cyclic double-error-correcting code which is given by its two roots. Line 9 in the decoding map $\mathcal{D}_{\mathcal{C}_{\text{DEC}}}$ is modified by substituting the decoding map of the new cyclic double-error-correcting code. The input to the modified decoding map $\mathcal{D}'_{\mathcal{C}_{\text{DEC}}}$ is the two roots of the cyclic double-error-correcting code, the n information cells, and the $2r$ redundancy cells, corresponding to the two syndromes of the two roots.

Decoding map $\mathcal{D}_{\mathcal{C}_{\text{TEC}}}$: The input is the memory-state vector $(\boldsymbol{c}', \boldsymbol{q}', \boldsymbol{p}_1', \boldsymbol{p}_2', \boldsymbol{p}_3')$. The output is the decoded data vector $\boldsymbol{v}$.

```
1. $s_1'' = \mathcal{D}_{\mathcal{C}_{WD}}(\boldsymbol{p}_1')$; $s_2'' = \mathcal{D}_{\mathcal{C}_{WD}}(\boldsymbol{p}_2')$; $s_3'' = \mathcal{D}_{\mathcal{C}_{WD}}(\boldsymbol{p}_3')$;
2. if ($s_1''$ == F)
3.    { $v = \mathcal{D}'_{\mathcal{C}_{\text{DEC}}}(\alpha_2, \alpha_3, c', p_2', p_3')$; return $v$; }
4. if ($s_2''$ == F)
5.    { $v = \mathcal{D}'_{\mathcal{C}_{\text{DEC}}}(\alpha_1, \alpha_3, c', p_1', p_3')$; return $v$; }
6. if ($s_3''$ == F)
7.    { $v = \mathcal{D}'_{\mathcal{C}_{\text{DEC}}}(\alpha_1, \alpha_2, c', p_1', p_2')$; return $v$; }
8. $s_1' = \sum_{i=0}^{n-1} c_i'\alpha_1^i$; $s_2' = \sum_{i=0}^{n-1} c_i'\alpha_2^i$; $s_3' = \sum_{i=0}^{n-1} c_i'\alpha_3^i$;
9. $e_1 = s_1' + s_1''$; $e_2 = s_2' + s_2''$; $e_3 = s_3' + s_3''$;
10. if (($\sum_{i=0}^{n-1} c_i'$) == ($\sum_{i=0}^{t-1} q_i'$))
11.    { $v = \mathcal{D}'_{\mathcal{C}_{\text{DEC}}}(\alpha_1, \alpha_2, c', p_1', p_2')$; return $v$; }
12. if (($e_1^{2^k+1}$ == $e_2$) OR ($e_1^{2^{3k}+1}$ == $e_3$)
                     OR ($e_2^{2^{2k}-2^k+1}$ == $e_3$))
13.    { $v$ = maj($\mathcal{D}'_{\mathcal{C}_{\text{SEC}}}(\alpha_1, c', p_1')$, $\mathcal{D}'_{\mathcal{C}_{\text{SEC}}}(\alpha_2, c', p_2')$,
                  $\mathcal{D}'_{\mathcal{C}_{\text{SEC}}}(\alpha_3, c', p_3')$); return $v$; }
14. $e' = \mathcal{D}_{\mathcal{C}_3^{\text{strong}}}(e_1, e_2, e_3)$;
15. $v = \mathcal{D}_{\mathcal{C}_W}(c' + e')$;
16. return $v$;
```

First, the three syndromes from the last three redundancy cell groups are decoded (line 1). If the decoded syndrome s_1'' is the error flag **F** (line 2), then there is at least one error in the group $\boldsymbol{p}_1'$. In the information cells $\boldsymbol{c}'$ and redundancy cells $\boldsymbol{p}_2', \boldsymbol{p}_3'$ there are at most two errors. Therefore, we decode by applying the decoding map $\mathcal{D}'_{\mathcal{C}_{\text{DEC}}}$ to $\boldsymbol{c}'$ and $\boldsymbol{p}_2', \boldsymbol{p}_3'$ with the roots α_1, α_3 (line 3). The same procedure is applied if s_2'' or s_3'' is the error flag **F** (lines $4-7$). Here, we use the property of $\mathcal{C}_3^{\text{strong}}$ that every two out of its three roots generate a cyclic double-error-correcting code.

After line 7, none of the syndromes s_1'', s_2'', s_3'' is the error flag **F**. Therefore, if there are errors in these redundancy cells then the number of errors in each of the three redundancy cells groups is even and since there are at most three errors, at most one group has exactly two errors. The received syndromes s_1', s_2', s_3' from the received cells and the differences e_1, e_2, e_3 are calculated (lines 8 and 9). If the condition in line 10 holds, then the cells $\boldsymbol{c}'$ and $\boldsymbol{q}'$ have zero or two errors. In both cases, the cells $\boldsymbol{c}', \boldsymbol{p}_1', \boldsymbol{p}_2'$ have at most two errors so it is possible to decode (line 11).

We are left with the case where the parities of the cells $\boldsymbol{c}'$ and $\boldsymbol{q}'$ are not the same. That is, these cells have either one or three errors. We address this case in the next lemma.

Lemma 4.6.9. *The condition in line* 12 *holds if and only if there is at most a single error in the information cells* $\boldsymbol{c}'$.

Proof. If there is at most a single error in the information cells $\boldsymbol{c}'$ then at most one of the redundancy cell groups $\boldsymbol{p}'_1, \boldsymbol{p}'_2, \boldsymbol{p}'_3$ has two errors, that is, at least two of these groups do not have errors. If there is no error in the first and second groups and the i-th information cell c'_i is in error, then $\boldsymbol{e}_1 = \alpha_1^i = \alpha^i$ and $\boldsymbol{e}_2 = \alpha_2^i = \alpha^{i(2^k+1)}$. Therefore, $\boldsymbol{e}_1^{2^k+1} = \boldsymbol{e}_2$. This condition clearly holds also if there are no errors in the information cells $\boldsymbol{c}'$. Similarly, if there is no error in $\boldsymbol{p}'_1$ and $\boldsymbol{p}'_3$ then $\boldsymbol{e}_1^{2^{3k}+1} = \boldsymbol{e}_3$, and if there is no error in $\boldsymbol{p}'_2$ and $\boldsymbol{p}'_3$ then $\boldsymbol{e}_2^{2^{2k}-2^k+1} = \boldsymbol{e}_3$. Therefore, if there is at most a single error in the information cells $\boldsymbol{c}'$ then the condition in line 12 holds.

Now assume that there is more than one error in the information cells $\boldsymbol{c}'$. That is, the information cells have two or three errors and in this case, there is no error in the redundancy cells $\boldsymbol{p}'_1, \boldsymbol{p}'_2, \boldsymbol{p}'_3$. Assume that the information cells have three errors in locations i, j, ℓ. Then,

$$\begin{aligned} e_1 &= \alpha^i + \alpha^j + \alpha^\ell \\ e_2 &= \alpha^{i(2^k+1)} + \alpha^{j(2^k+1)} + \alpha^{\ell(2^k+1)} \\ e_3 &= \alpha^{i(2^{3k}+1)} + \alpha^{j(2^{3k}+1)} + \alpha^{\ell(2^{3k}+1)}, \end{aligned}$$

for some $0 \leqslant i < j < \ell \leqslant n-1$. In this case, $\boldsymbol{e}_1^{2^k+1} \neq \boldsymbol{e}_2$. Otherwise, we get

$$\begin{aligned} &e_1 + \alpha^i + \alpha^j + \alpha^\ell = 0 \\ &e_1^{(2^k+1)} + \alpha^{i(2^k+1)} + \alpha^{j(2^k+1)} + \alpha^{\ell(2^k+1)} = 0, \end{aligned}$$

and $\mathcal{C}(\alpha, \alpha^{2^k+1})$ has a codeword of weight at most four, which is a contradiction. Similarly, $\boldsymbol{e}_1^{2^{3k}+1} \neq \boldsymbol{e}_3$ and $\boldsymbol{e}_2^{2^{2k}-2^k+1} \neq \boldsymbol{e}_3$. The case of two errors in the information cells is handled similarly. Hence, the condition in line 12 does not hold. ■

According to Lemma 4.6.9, if the condition in line 12 holds, then there is at most a single error in the information cells $\boldsymbol{c}'$. At most one of the redundancy cell groups $\boldsymbol{p}'_1, \boldsymbol{p}'_2, \boldsymbol{p}'_3$ has errors. Therefore, at least two out of the three decoding maps in line 13 succeed, and the function **`maj`**, which outputs the majority of the three decoded values, returns the correct value of $\boldsymbol{v}$. In line 14, there are at most three errors in the information cells and no errors in the redundancy cell groups $\boldsymbol{p}'_1, \boldsymbol{p}'_2, \boldsymbol{p}'_3$, so it is possible to find the error vector (line 14) and decode (line 15). We conclude with the following theorem.

Theorem 4.6.10. *If $\mathcal{C}_{\mathcal{W}}$ is an $[n, k, t]$ WOM-code, $\mathcal{C}_{\mathcal{WD}}$ is an $[r, \lceil \log_2(n+1) \rceil, t]$ single-error-detecting WOM-code, and $\gcd(\lceil \log_2(n+1) \rceil, 6) = 1$, then $\mathcal{C}_{TEC}$ is an $[n + 3r + t, k, t]$ triple-error-correcting WOM-code.*

4.7 Multiple Error-Correcting WOM-Codes

In this section, we study how to correct an arbitrary number of errors with a WOM-code. As described in the Introduction, a simple scheme to construct an e-error-correcting WOM-code is done by using an existing WOM-code and replicating each one of its cells $2e + 1$ times. A first improvement upon this scheme can be achieved by replicating each cell only $e + 1$ times. Then, instead of using a regular WOM-code, a single-error-correcting WOM-code is applied. Note that since each cell is replicated $e + 1$ times, at most one cell in the single-error-correcting WOM-code will be erroneous and thus its decoding map succeeds. In the rest of the section we will show how to use similar ideas in order to construct better WOM-codes that correct an arbitrary specified number of errors.

Let us first show another property of the triple-error-correcting WOM-code studied in Section 4.6.

Lemma 4.7.1. *Let $\mathcal{C}_{TEC}$ be an $[n + 3r + t, k, t]$ triple-error-correcting WOM-code constructed in Theorem* 4.6.10. *Then the code $\mathcal{C}_{TEC}$ can correct four erasures.*

Proof. Assume first that there are no erasures in the redundancy cells groups $\boldsymbol{p}_1, \boldsymbol{p}_2, \boldsymbol{p}_3$ then we know the correct value of the syndromes s_1, s_2, s_3 and in the information cells $\boldsymbol{c}$ there are at most four erasures. Since the code $\mathcal{C}_3^{\text{strong}}$ corrects three errors, its minimum distance is at least seven and hence it can correct up to six erasures and a fortiori four erasures.

If each redundancy cell group $\boldsymbol{p}_1, \boldsymbol{p}_2, \boldsymbol{p}_3$ has at most one error, then it is still possible to successfully decode the three syndromes since each syndrome is stored using a single-error-detecting WOM-code and then find the erasure cells as in the first case.

If one of the three redundancy groups has at least two erasures then in the n information cells and two other redundancy groups there are at most two erasures and again it is possible to successfully decode the erasure values. ■

The next theorem confirms the validity of the first construction for an e-error-correcting WOM-code.

Theorem 4.7.2. *Let $\mathcal{C}_{\text{TEC}}$ be an $[n,k,t]$ triple-error-correcting WOM-code. Then there exists an $[\lceil e/2 \rceil n, k, t]$ e-error-correcting WOM-code.*

Proof. Let us denote the cells of the WOM-code $\mathcal{C}_{\text{TEC}}$ by $c'_0, \ldots, c'_{n-1}$. The constructed e-error-correcting WOM-code is denoted by $\mathcal{C}_{\text{eEC}}$ and its $\lceil e/2 \rceil n$ cells are denoted by $c_{0,0}, \ldots, c_{0,\lceil e/2 \rceil -1}, \ldots, c_{n-1,0}, \ldots, c_{n-1,\lceil e/2 \rceil -1}$. We use two transformations in the validation of the construction. The first transformation

$$f : \{0,1\}^{\lceil e/2 \rceil n} \to \{0,1,?\}^n,$$

transforms a memory-state vector of $\lceil e/2 \rceil n$ cells,

$$\boldsymbol{c} = (c_{0,0}, \ldots, c_{0,\lceil e/2 \rceil -1}, \ldots, c_{n-1,0}, \ldots, c_{n-1,\lceil e/2 \rceil -1}),$$

into a memory-state vector of n cells,

$$\boldsymbol{c}' = (c'_0, \ldots, c'_{n-1}),$$

by taking the majority of every group of $\lceil e/2 \rceil$ cells. That is, for all $0 \leqslant i \leqslant n-1$

$$c'_i = \text{maj}\{c_{i,0}, \ldots, c_{i,\lceil e/2 \rceil -1}\},$$

and in case of equality in the numbers of ones and zeros, then $c'_i = ?$, the erasure symbol. The second transformation

$$g : \{0,1\}^n \to \{0,1\}^{\lceil e/2 \rceil n},$$

transforms a memory-state vector of n cells,

$$\boldsymbol{c}' = (c'_0, \ldots, c'_{n-1}),$$

to a memory-state vector of $\lceil e/2 \rceil n$ cells,

$$\boldsymbol{c} = (c_{0,0}, \ldots, c_{0,\lceil e/2 \rceil -1}, \ldots, c_{n-1,0}, \ldots, c_{n-1,\lceil e/2 \rceil -1}),$$

such that for all $0 \leqslant i \leqslant n-1$ and $0 \leqslant j \leqslant \lceil e/2 \rceil - 1$,

$$c_{i,j} = c'_i.$$

That is, every cell is replicated $\lceil e/2 \rceil$ times.

In the encoding map $\mathcal{E}_{\mathcal{C}_{\text{eEC}}}$, the new vector data $\boldsymbol{v}$ and memory-state vector $\boldsymbol{c}$ of $\lceil e/2 \rceil n$ cells are received. Then, the new memory-state vector is updated according to

$$g(\mathcal{E}_{\mathcal{C}_{\text{TEC}}}(f(\boldsymbol{c}), \boldsymbol{v})).$$

First, a memory-state vector of n cells is generated by the transformation f on the memory-state vector of the $\lceil e/2 \rceil n$ cells, $\boldsymbol{c}$. Then, the encoding map $\mathcal{E}_{\mathcal{C}_{\text{TEC}}}$ is invoked on the memory-state vector $f(\boldsymbol{c})$ and data vector $\boldsymbol{v}$. Finally, the new memory-state vector of n cells is transformed back to $\lceil e/2 \rceil n$ cells to generate the new memory-state vector.

In the decoding map $\mathcal{E}_{\mathcal{D}_{\text{eEC}}}$, the memory-state vector $\boldsymbol{c}$ of $\lceil e/2 \rceil n$ cells is the input and is decoded according to

$$\mathcal{E}_{\mathcal{D}_{\text{TEC}}}(f(\boldsymbol{c})).$$

As in the encoding map, first a memory-state vector of n cells is generated from the memory-state vector of $\lceil e/2 \rceil n$ cells and is the input to the decoding map of the WOM-code $\mathcal{E}_{\mathcal{D}_{\text{TEC}}}$. The output data vector $\boldsymbol{v}$ from $\mathcal{E}_{\mathcal{D}_{\text{TEC}}}$ is the output data vector of the decoding map.

If there are at most e errors in $\boldsymbol{c}$ then in the memory-state vector $f(\boldsymbol{c})$ there are at most three errors and erasures or exactly four erasures. Since $\mathcal{C}_{\text{TEC}}$ is a triple-error-correcting WOM-code it can correct three errors and erasures and according to Lemma 4.7.1 it can correct four erasures as well. ■

The next example demonstrates how to use the previous construction in order to construct a four-error-correcting WOM-code.

Example 4.7.1. Let us start with the $[2^k - 1, k, 5 \cdot 2^{k-4} + 1]$ WOM-code for $k \geqslant 4$, and $\gcd(k, 6) = 1$, by Godlewski [5]. First, a strong triple-error-correcting code exists since we require that $\gcd(k, 6) = 1$. A triple-error-correcting WOM-code is built using the $[2^k, k, 5 \cdot 2^{k-4} + 1]$ single-error-detecting WOM-code from Example 4.3.1. This last WOM-code is used as the WOM-codes $\mathcal{C}_{\mathcal{WD}}$ and $\mathcal{C}'_{\mathcal{W}}$ in the construction of the triple-error-correcting WOM-codes. Hence, we get a triple-error-correcting WOM-code that stores k bits $5 \cdot 2^{k-4} + 1$ times using

$$2^k - 1 + 1 + 3 \cdot 2^k = 4 \cdot 2^k$$

cells. Then, according to Theorem 4.7.2 there exists a $[8 \cdot 2^k, k, 5 \cdot 2^{k-4} + 1]$ four-error-correcting WOM-code.

Roth [15] suggested another construction of multiple-error-correcting WOM-codes. The construction is based upon a recursive approach, described as follows. Assume that $\mathcal{C}$ is an $[n, k, t]$ WOM-code, and assume that there exists a linear e-error-correcting code of length n and redundancy r. Then, the r redundancy bits are recursively stored using another e-error-correcting WOM-code. This process can be recursively repeated multiple-times until it is necessary to use an e-error-correcting WOM-code which can be constructed according to Theorem 4.7.2. We validate the recursive step of this construction in the next theorem and then show an example of how to use the construction.

Theorem 4.7.3. *Let $\mathcal{C}_1$ be an $[n, k, t]$ WOM-code, $\mathcal{C}_2$ be a linear e-error-correcting-code of length n and redundancy r, and $\mathcal{C}_3$ be an $[m, r, t]$ e-error-correcting WOM-code, then there exists an $[n+m, k, t]$ e-error-correcting WOM-code.*

Proof. The e-error-correcting WOM-code we construct has $n+m$ cells which are partitioned into two groups. The first group has n cells and is denoted by $\boldsymbol{c} = (c_1, \ldots, c_n)$. The second group consists of m cells and is denoted by $\boldsymbol{p} = (p_1, \ldots, p_m)$.

In the encoding map the memory-state vector of $n+m$ cells, $(\boldsymbol{c}, \boldsymbol{p})$ and new data vector $\boldsymbol{v}$ are received. The output is a new memory-state vector $(\boldsymbol{c}', \boldsymbol{p}')$. The data vector $\boldsymbol{v}$ is stored in the first n cells using the encoding map of the WOM-code $\mathcal{C}_1$,

$$\boldsymbol{c}' = \mathcal{E}_{\mathcal{C}_1}(\boldsymbol{c}, \boldsymbol{v}).$$

Let H be the parity check matrix of the linear e-error-correcting code $\mathcal{C}_2$. In the next step a syndrome $\boldsymbol{s}$ of r bits is calculated using the new value of the n bits,

$$\boldsymbol{s} = H \cdot \boldsymbol{c}'.$$

Then, the syndrome $\boldsymbol{s}$ is stored in the m cells using the encoding map of the WOM-code $\mathcal{C}_3$,

$$\boldsymbol{p}' = \mathcal{E}_{\mathcal{C}_3}(\boldsymbol{p}, \boldsymbol{s}).$$

In the decoding map, the memory-state vector $(\boldsymbol{c}', \boldsymbol{p}')$ is the input and the output is a data vector $\boldsymbol{v}$ of k bits. First, the syndrome $\boldsymbol{s}$ of r bits is decoded by applying the decoding map of the e-error-correcting WOM-code $\mathcal{C}_3$,

$$\boldsymbol{s}'' = \mathcal{D}_{\mathcal{C}_3}(\boldsymbol{p}').$$

The success of this decoding map is guaranteed since there are at most e errors in $\boldsymbol{p}'$ and the WOM-code $\mathcal{C}_3$ can correct e errors. Another syndrome is calculated from the n cells and the parity check matrix H,

$$s' = H \cdot c'.$$

Note that if the memory-state vector without errors is $\boldsymbol{c}$ and $\boldsymbol{e}$ is the error-vector of weight at most e, i.e. $\boldsymbol{c}' = \boldsymbol{c} + \boldsymbol{e}$, then

$$H \cdot \boldsymbol{e} = H \cdot (\boldsymbol{c} + \boldsymbol{c}') = H \cdot \boldsymbol{c} + H \cdot \boldsymbol{c}' = \boldsymbol{s}'' + \boldsymbol{s}'.$$

Therefore, the syndrome that corresponds to the error vector $\boldsymbol{e}$ is $\boldsymbol{s}'' + \boldsymbol{s}'$ and it is possible to find it by applying the decoding map of the code $\mathcal{C}_2$ to $\boldsymbol{s}'' + \boldsymbol{s}'$,

$$\boldsymbol{e} = \mathcal{D}_{\mathcal{C}_2}(\boldsymbol{s}'' + \boldsymbol{s}').$$

Finally, the data vector $\boldsymbol{v}$ is decoded by applying the decoding map of the WOM-code $\mathcal{C}_1$ to the memory-state vector $\boldsymbol{c}' + \boldsymbol{e}$,

$$\boldsymbol{v} = \mathcal{D}_{\mathcal{C}_1}(\boldsymbol{c}' + \boldsymbol{e}).$$

■

A necessary condition to efficiently apply this scheme recursively is that, r, the number of redundancy bits of the e-error-correcting is not greater than the number of information bits k; otherwise the number of cells in the next step of the recursion is greater than the total number of cells n. The code constructed in Example 4.7.1 cannot be used for just this reason. If we start with the $[2^k - 1, k, 5 \cdot 2^{k-4} + 1]$ WOM-code for $k \geqslant 4, \gcd(k, 6) = 1$, and then use a four-error-correcting-code, the number of redundancy bits is roughly $4k$ and so the number of information bits for the next WOM-code in the recursion is greater than the number of the information bits that the WOM-code needs to store. The next example shows another case where this scheme can outperform the construction in Theorem 4.7.2.

Example 4.7.2. In this example we start with the $[23, 11, 3]$ WOM-code constructed by Cohen et al. [2]. In order to use this WOM-code in a larger block of cells, one can simply repeat the WOM-code in successive groups of 23 cells. For example, repeating the code 89 times provides us with a $[2047, 979, 3]$ WOM-code. In order to construct a four-error-correcting WOM-code according to the construction in Theorem 4.7.2, it is necessary to first build a triple-error-correcting WOM-code. In this case $n = 2047$, $\lceil \log(n + 1) \rceil = 11$, and we will construct a single-error-detecting WOM-code that stores 11 bits three times. This can be done according to Section 4.3 and the $[23, 11, 3]$ WOM-code, so we receive a $[26, 11, 3]$ single-error-detecting

WOM-code. The condition of Theorem 4.6.10 holds, i.e. $\gcd(11,6) = 1$, and thus we can construct a $[2047+3\cdot 26+3 = 2128, 979, 3]$ triple-error-correcting WOM-code. Finally, by applying Theorem 4.7.2, we can construct a $[4256, 979, 3]$ four-error-correcting WOM-code.

Next, we construct the code according to Theorem 4.7.3. Again, let us start with the $[2047, 979, 3]$ WOM-code and use a four-error-correcting code of length 2047. Specifically, we use a four-error-correcting BCH code of $4\cdot 11 = 44$ redundancy bits, so we need to store 44 bits three times while correcting four errors. Therefore, we seek to use Theorem 4.7.2 and hence need to construct first a triple-error-correcting WOM-code which stores 44 bits three times. Note that now $n = 92$ and $\lceil \log(n+1) \rceil = 7$, so a single-error-detecting that stores seven bits three times is required. Cohen et al. [2] also constructed a $[7, 3, 3]$ WOM-code and therefore there exists a $[14, 6, 3]$ WOM-code. By simply adding three more cells to store one more bit three times we construct a $[17, 7, 3]$ WOM-code. The latter WOM-code provides us with a $[20, 7, 3]$ single-error-detecting WOM-code. The condition of Theorem 4.6.10 holds again, $\gcd(7,6) = 1$, and thus we construct a $[92+3\cdot 20+3 = 155, 44, 3]$ triple-error-correcting WOM-code. Next, by applying Theorem 4.7.2, we can construct a $[2\cdot 155 = 310, 44, 3]$ four-error-correcting WOM-code. Finally, we get a $[2047+310 = 2357, 979, 3]$ four-error-correcting WOM-code, thereby improving upon the first construction.

4.8 Summary and Conclusions

In this chapter, we constructed error-correcting WOM-codes. All the proposed constructions had the same structure in the sense that we started with an existing $[n, k, t]$ WOM-code and then added more redundancy cells that enabled the WOM-code to detect or correct errors. We started with a construction of a single-error-detecting WOM-code. Then, we showed how to use this construction along with Hamming codes in order to construct single-error-correcting WOM-codes. Building upon this, we construct double-error-correcting WOM-codes when $\lceil \log(n+1) \rceil$ is an odd integer and when $\lceil \log(n+2) \rceil$ is an even integer.

We proceeded to construct triple-error-correcting WOM-codes. Here, we introduced the notion of strong cyclic error-correcting codes, for which the e roots of the generator polynomial have the property that any subset of k distinct roots generate a k-error-correcting code. We showed how to find strong triple-error correcting codes and used them in the construction of triple-error-correcting WOM-codes. The triple-error-correcting WOM-codes were used in one construction, which then formed the basis of a recursive construction that sometimes yields better multiple-error-correcting WOM-codes.

Acknowledgment

Chapter 4 is in part a reprint of the material in the paper: E. Yaakobi, P.H. Siegel, A. Vardy, and J.K. Wolf, "Multiple Error-Correcting WOM-Codes," *Proc. IEEE International Symposium on Information Theory*, pp. 1933–1937, Austin, Texas, June 2010.

Bibliography

[1] C. Bracken and T. Helleseth, "Triple-error-correcting BCH-like codes," in *Proc. IEEE Int. Symp. Inform. Theory*, pp. 1723–1725, Seoul, Korea, June 2009.

[2] G.D. Cohen, P. Godlewski, and F. Merkx, "Linear binary code for write-once memories," *IEEE Trans. Inform. Theory*, vol. 32, no. 5, pp. 697–700, September 1986.

[3] A. Fiat and A. Shamir, "Generalized write-once memories," *IEEE Trans. Inform. Theory*, vol. 30, pp. 470–480, September 1984.

[4] F. Fu and A.J. Han Vinck, "On the capacity of generalized write-once memory with state transitions described by an arbitrary directed acyclic graph," *IEEE Trans. Inform. Theory*, vol. 45, no. 1, pp. 308–313, September 1999.

[5] P. Godlewski, "WOM-codes construits à partir des codes de Hamming," *Discrete Math.*, no. 65 pp. 237–243, 1987.

[6] C. Heegard, "On the capacity of permanent memory," *IEEE Trans. Inform. Theory*, vol. 31, no. 1, pp. 34–42, January 1985.

[7] A. Jiang, "On the generalization of error-correcting WOM codes," in *Proc. IEEE Int. Symp. Inform. Theory*, pp. 1391–1395, Nice, France, June 2007.

[8] A. Jiang, V. Bohossian, and J. Bruck, "Floating codes for joint information storage in write asymmetric memories," in *Proc. IEEE Int. Symp. Inform. Theory*, pp. 1166–1170, Nice, France, June 2007.

[9] A. Jiang and J. Bruck, "Joint coding for flash memory storage," in *Proc. IEEE Int. Symp. Inform. Theory*, pp. 1741–1745, Toronto, Canada, July 2008.

[10] T. Kasami, "The weight enumerators for several classes of subcodes of the second order binary Reed-Muller codes," *Inform. and Control*, vol. 18, pp. 369–394, September 1971.

[11] S. Kayser, E. Yaakobi, P.H. Siegel, A. Vardy, and J.K. Wolf, "Multiple-write WOM-codes," in *48-th Annual Allerton Conference on Communication, Control and Computing*, Monticello, IL, September 2010.

[12] H. Mahdavifar, P.H. Siegel, A. Vardy, J.K. Wolf, and E. Yaakobi, "A nearly optimal construction of flash codes," in *Proc. IEEE Int. Symp. Inform. Theory*, pp. 1239–1243, Seoul, Korea, July 2009.

[13] F. Merkx, “Womcodes constructed with projective geometries,” *Traitement du signal*, vol. 1, no. 2-2, pp. 227–231, 1984.

[14] R.L. Rivest and A. Shamir, “How to reuse a write-once memory,” *Inform. and Control*, vol. 55, nos. 1–3, pp. 1–19, December 1982.

[15] R.M. Roth, February 2010, personal communication.

[16] J.K. Wolf, A.D. Wyner, J. Ziv, and J. Korner, “Coding for a write-once memory,” *AT&T Bell Labs. Tech. J.*, vol. 63, no. 6, pp. 1089–1112, 1984.

[17] Y. Wu, “Low complexity codes for writing write-once memory twice,” in *Proc. IEEE Int. Symp. Inform. Theory*, pp. 1928–1932, Austin, Texas, June 2010.

[18] Y. Wu and A. Jiang, “Position modulation code for rewriting write-once memories,” submitted to *IEEE Trans. Inform. Theory*, September 2009. Preprint available at http://arxiv.org/abs/1001.0167.

[19] E. Yaakobi, S. Kayser, P.H. Siegel, A. Vardy, and J.K. Wolf, “Efficient two-write WOM-codes,” in *Proc. IEEE Inform. Theory Workshop*, Dublin, Ireland, August 2010.

[20] G. Zémor and G.D. Cohen, “Error-correcting WOM-codes,” *IEEE Trans. Inform. Theory*, vol. 37, no. 3, pp. 730–734, May 1991.

[21] G. Zémor, “Problèmes combinatoires liés à l’écriture sur des mémoires,” Ph.D. Dissertation, ENST, Paris, France, November 1989.

Chapter 5

On Codes that Correct Asymmetric Errors with Graded Magnitude Distribution

5.1 introduction

The topic of asymmetric error-correcting codes over non-binary alphabets has attracted considerable attention in the past few years, largely due to its relevance in the context of multi-level flash memories. However, research on asymmetric codes has a long history. A number of papers appeared in the 1960's, e.g., [3, 13, 19, 20]. Constructions and upper bounds on such codes were given in, e.g., [2,7,8,11,12,15,21] and constructions of systematic asymmetric error-correcting codes were studied in [4]. One of the dominant error mechanisms of flash memory cells is over-programming the cells [6,17,22]. Since flash memory cells cannot reduce their level, these errors cannot be physically corrected unless the entire containing block is erased. Thus, it is crucial to design error-correcting codes that correct asymmetric errors of limited-magnitude. Furthermore, the ability to correct such errors can enable the programming of the cells to be less accurate and thus faster.

In [5], Cassuto et al. designed codes which correct t asymmetric errors of limited-magnitude ℓ. In this model, an error can only increase the erroneous symbol by at most ℓ levels. Systematic optimal codes for this model that correct all asymmetric and symmetric errors of limited-magnitude were given by Elarief and Bose [10]. In [16], the case of correcting a single asymmetric error ($t = 1$) of limited-magnitude ℓ was studied, and the results improved upon those given by Cassuto et al. for this scenario. Asymmetric error-correcting codes for binary and non-binary alphabets were recently presented by Dolecek [9]. Codes correcting all unidirectional errors of limited-magnitude were studied in [1]. Another related error model assumes that if the cell level is x then the level can only be reduced to any value less than x. Code constructions were given in [14], and a short survey was given in [15].

These previously proposed codes and bounds for the non-binary case mainly deal with the case of t asymmetric errors of limited-magnitude ℓ. However, in flash memories, it is likely that only a few cells will suffer from an error of large magnitude and that most of the erroneous cells will suffer from an error of a smaller magnitude [22]. In this chaptr, we will present code constructions that correct t_1 asymmetric errors of magnitude at most ℓ_1 and t_2 asymmetric errors of magnitude at most ℓ_2, where $\ell_1 < \ell_2$. This model can be naturally generalized to a wider range of magnitudes as well as for errors in both directions.

5.2 Preliminaries

In this work, the memory elements, called cells, have q states: $0, 1, \ldots, q-1$. For a vector $\boldsymbol{x} = (x_1, \ldots, x_n)$, we let $wt(\boldsymbol{x})$ denote its Hamming weight, i.e. $wt(\boldsymbol{x}) = |\{i \mid x_i \neq 0\}|$. First, let us define asymmetric limited-magnitude errors.

Definition 5.2.1. *An error vector $\boldsymbol{e} = (e_1, e_2, \ldots, e_n)$ is called a **t-asymmetric ℓ-limited-magnitude error** if*

1. $\max_{1 \leqslant i \leqslant n}\{e_i\} \leqslant \ell$,
2. $wt(\boldsymbol{e}) \leqslant t$.

*An $[n, q, t, \ell]$ error-correcting code $\mathcal{C}$ is called a **t-asymmetric ℓ-limited-magnitude error-correcting code** if it is a q-ary code of length n which can correct all t-asymmetric ℓ-limited-magnitude errors.*

We extend the last definition to error vectors with two different limited-magnitudes.

Definition 5.2.2. *An error vector $\boldsymbol{e} = (e_1, e_1, \ldots, e_n)$ is called a (t_1, t_2)**-asymmetric** (ℓ_1, ℓ_2)**-limited-magnitude error** if*

1. $\max_{1 \leqslant i \leqslant n}\{e_i\} \leqslant \ell_2$.
2. $wt(\boldsymbol{e}) \leqslant t_1 + t_2$.

3. $|\{i \mid \ell_1 + 1 \leqslant e_i \leqslant \ell_2\}| \leqslant t_2$,

An $[n, q, (t_1, t_2), (\ell_1, \ell_2)]$ *error-correcting code* $\mathcal{C}$ *is called a* $\boldsymbol{(t_1, t_2)}$***-asymmetric*** $\boldsymbol{(\ell_1, \ell_2)}$***-limited-magnitude error-correcting code*** *if it is a* q*-ary code of length* n *which can correct all* (t_1, t_2)*-asymmetric* (ℓ_1, ℓ_2)*-limited-magnitude errors.*

That is, the error model is such that there are at most $t_1 + t_2$ errors; at most t_2 of these errors have magnitude between $\ell_1 + 1$ and ℓ_2 and the magnitude of the rest of the errors is at most ℓ_1.

Lemma 5.2.3. *Let* t_1, t_2, ℓ_1, ℓ_2 *be positive integers such that* $\ell_1 < \ell_2$*. Then, the number of* (t_1, t_2)*-asymmetric* (ℓ_1, ℓ_2)*-limited-magnitude errors is*

$$\sum_{i=0}^{t_2} \left(\binom{n}{i} (\ell_2 - \ell_1)^i \cdot \sum_{j=0}^{t_1+t_2-i} \binom{n-i}{j} \ell_1^j \right).$$

Proof. For any (t_1, t_2)-asymmetric (ℓ_1, ℓ_2)-limited-magnitude error vector, the number of errors of magnitude between $\ell_1 + 1$ and ℓ_2 is at most t_2. Assume this number is i, $0 \leqslant i \leqslant t_2$, then the number of error vectors with i such errors is $\binom{n}{i}(\ell_2 - \ell_1)^i$. There are at most $t_1 + t_2 - i$ more errors of magnitude at most ℓ_1 and so for any error vector with i errors between $\ell_1 + 1$ and ℓ_2, the number of (t_1, t_2)-asymmetric (ℓ_1, ℓ_2)-limited-magnitude error vectors is $\sum_{j=0}^{t_1+t_2-i} \binom{n-i}{j} \ell_1^j$. Therefore, the total number of such error vectors is $\sum_{i=0}^{t_2} \left(\binom{n}{i} (\ell_2 - \ell_1)^i \cdot \sum_{j=0}^{t_1+t_2-i} \binom{n-i}{j} \ell_1^j \right)$. ■

There are two error models that can be considered. The errors can or cannot wrap-around. That is, in the first case, if the transmitted word is $\boldsymbol{c}$ and the error vector is $\boldsymbol{e}$ then the received word is $(\boldsymbol{c} + \boldsymbol{e}) \mod q$, while in the latter case we require that $\boldsymbol{c} + \boldsymbol{e} \leqslant (q-1, \ldots, q-1)$. In many practical applications like multi-level flash memories, it is common to assume that errors do not wrap-around. However, the constructions we present can work in some cases for both models.

5.3 Constructions of t-Asymmetric ℓ-Limited-Magnitude Error-Correcting Codes

The goal of this work is to construct (t_1, t_2)-asymmetric (ℓ_1, ℓ_2)-limited-magnitude error-correcting codes. The construction of such codes is based on a recent construction by Cassuto et al. [5] of t-asymmetric ℓ-limited magnitude error-correcting codes. We now review the construction in [5]. For a vector $\boldsymbol{x} = (x_1, \ldots, x_n)$, and a positive integer m, we define the vector $\boldsymbol{x} \bmod m$ to be

$$\boldsymbol{x} \bmod m = (x_1 \bmod m, \ldots, x_n \bmod m).$$

Construction 5.3.1. Let Σ be a t-error-correcting code of size n and redundancy r over an alphabet of size $\ell + 1$. Then the q-ary code $\mathcal{C}$ of length n is defined as

$$\mathcal{C} = \{\boldsymbol{c} \in \{0, \ldots, q-1\}^n \mid \boldsymbol{c} \bmod (\ell+1) \in \Sigma\}.$$

The code Σ will be called the *base code* used to construct $\mathcal{C}$. The following theorem was proved in [5].

Theorem 5.3.1. *The code $\mathcal{C}$ is an $[n, q, t, \ell]$ error-correcting code if the code Σ corrects t or fewer symmetric errors. If $q > 2\ell$, the converse is true as well.*

Next, we describe the decoding and encoding procedures.

Decoding: Let $\boldsymbol{c} \in \mathcal{C}$ be the transmitted codeword and $\boldsymbol{y} = \boldsymbol{c} + \boldsymbol{e}$ the received word, where $\boldsymbol{e}$ is a t-asymmetric ℓ-limited-magnitude error vector. Let

$$\boldsymbol{z} = \boldsymbol{y} \bmod (\ell+1) = (\boldsymbol{c} + \boldsymbol{e}) \bmod (\ell+1).$$

Then, since $\boldsymbol{c} \bmod (\ell+1) \in \Sigma$, the word $\boldsymbol{z}$ suffers at most t symbol errors. These errors can be found using the decoder of the code Σ. That is, the value of $\boldsymbol{e} \bmod (\ell+1)$ is found and thus also the error vector $\boldsymbol{e}$.

Remark 5.3.1. As mentioned in [5], we will also assume here, for the simplicity of the encoding procedure, that $(\ell+1)|q$, and the construction corrects wrap-around errors as well. However it is possible to modify the encoding procedure also for the case where $(\ell+1) \nmid q$, while sacrificing the ability to correct wrap-around errors.

Encoding: The encoding procedure, as presented in [5], can use any encoding procedure for Σ. However, for our construction, we will require that Σ be systematic[1]. If r is the redundancy of Σ then the encoder's input is a vector $(\boldsymbol{u}_1, \boldsymbol{u}_2) \in \{0, \ldots, q-1\}^{n-r} \times \{0, \ldots, \frac{q}{\ell+1} - 1\}^r$. Let $\boldsymbol{v}_1 \in \{0, \ldots, \ell\}^r$ be the systematic encoder's output of Σ when applied to $(\boldsymbol{u}_1 \bmod (\ell+1))$. Then the encoder's output of $\mathcal{C}$ is $\boldsymbol{c}$, where

$$\boldsymbol{c} = (\boldsymbol{u}_1, (\ell+1) \cdot \boldsymbol{u}_2 + \boldsymbol{v}_1).$$

[1]The restriction that the code Σ has a systematic encoder is not a severe one, as many codes and in particular all linear codes have a systematic encoder.

Note that $\boldsymbol{c} \in \{0, \ldots, q-1\}^n$, $(\boldsymbol{c} \bmod (\ell+1)) \in \Sigma$ and distinct input vectors generate distinct output vectors.

In the rest of this chapter, we present code constructions that are based on the codes we just described. When we refer to an $[n, q, t, \ell]$ code $\mathcal{C}$, we refer to a code that is designed in Construction 5.3.1 which is constructed using a base code Σ. While Σ is constructed over an alphabet of size $\ell + 1$ and has to correct t symbol errors, it is possible to use other codes over larger alphabets that correct t-asymmetric ℓ-limited-magnitude errors that wrap around (see Construction 1A in [5]). Either choice of Σ will work in our constructions.

In fact, assume one wants to construct $[n, q, t, 1]$ error-correcting codes. According to Construction 5.3.1, Σ is a binary code, however if q is an odd integer the construction does not necessarily result in a good code. A different construction of $[n, q, t, 1]$ error-correcting codes was recently given by Dolecek [9]. Yet another construction is presented in the next theorem and provides the code Σ to be used as an $[n, p, t, 1]$ error-correcting code in order to construct an $[n, q, t, 1]$ error-correcting code where p is a prime integer that divides q.

Theorem 5.3.2. *Let p, t, m, n be four positive integers such that p is a prime number, $t \leqslant p-1$, and $n = p^m - 1$. Let $\alpha \in GF(p^m)$ be a primitive element. Then, the matrix H,*

$$H = \begin{pmatrix} \alpha^1 & \alpha^2 & \alpha^3 & \cdots & \alpha^n \\ \alpha^2 & \alpha^4 & \alpha^6 & \cdots & \alpha^{2n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \alpha^t & \alpha^{2t} & \alpha^{3t} & \cdots & \alpha^{tn} \end{pmatrix},$$

is a parity-check matrix of an $[n, p, t, 1]$ error-correcting code of dimension $m - t$ over $GF(p)$.

Proof. Assume that there are t errors in locations $i_1, \ldots, i_t$ and the error magnitude in each coordinate is one. Therefore, the syndrome we receive is $\boldsymbol{s} = (s_1, \ldots, s_t)$ where for $1 \leqslant \ell \leqslant t$,

$$s_\ell = \sum_{k=1}^{t} \alpha_k^\ell.$$

According to the Newton-Girard formulas over $GF(p)$ [18] and since $t \leqslant p - 1$, it is possible for $1 \leqslant \ell \leqslant t$ to derive the values of

$$\Lambda_\ell = \sum_{v_1 < v_2 < \cdots < v_\ell} \alpha_{v_1} \cdot \alpha_{v_2} \cdots \alpha_{v_\ell}.$$

These values give us the coefficients of the degree-t polynomial $p(x) = \prod_{j=1}^{t}(x - \alpha_{i_j})$, since

$$p(x) = x^t - \Lambda_1 x^{t-1} + \Lambda_2 x^{t-2} - \cdots + (-1)^{t-1}\Lambda_{t-1}x + (-1)^t \Lambda_t.$$

Therefore, the roots of the polynomial $p(x)$ are the values $\alpha_{i_1}, \ldots, \alpha_{i_t}$ and we can derive the error locations.

Since the number of errors is not known in advance, we can perform the same rule of decoding while some of the roots will have the value zero which correspond to no error. ■

Before we proceed to the next section and construct (t_1, t_2)-asymmetric (ℓ_1, ℓ_2)-limited-magnitude error-correcting codes, we construct a family of codes that correct errors of the following magnitudes:

$$s, 2s, 3s, \ldots, \ell s,$$

for some positive integers s, ℓ.

Definition 5.3.3. *An error vector* $\boldsymbol{e} = (e_1, \ldots, e_n)$ *is called a* ***t-asymmetric*** $(\boldsymbol{\ell}, \boldsymbol{s})$***-multiple-spaced limited-magnitude error*** *if*

1. $\max_{1 \leqslant i \leqslant n}\{e_i\} \leqslant \ell s$,
2. $wt(\boldsymbol{e}) \leqslant t$,
3. *for all* $1 \leqslant i \leqslant n$, $e_i \equiv 0 (\bmod s)$.

An $[n, q, t, \ell, s]$ *error-correcting-code* $\mathcal{C}$ *is called a* ***t-asymmetric*** $(\boldsymbol{\ell}, \boldsymbol{s})$***-multiple-spaced limited-magnitude error-correcting code*** *if it is a* q*-ary code of length* n *which can correct all* t*-asymmetric* (ℓ, s)*-multiple-spaced limited-magnitude errors.*

The next theorem gives a construction of $[n, q, t, \ell, s]$ error-correcting-codes.

Theorem 5.3.4. *Let* n, q, t, ℓ, s *be positive integers and assume that there exists an* $[n, \lceil \frac{q}{s} \rceil, t, \ell]$ *error-correcting code* $\mathcal{C}_1$*. Then, there exists an* $[n, q, t, \ell, s]$ *error-correcting code* $\mathcal{C}_2$ *of the same size.*

Proof. The new code $\mathcal{C}_2$ is defined as follows.

$$\boldsymbol{c} \in \mathcal{C}_2 \text{ if and only if } \left\lfloor \frac{1}{s} \cdot \boldsymbol{c} \right\rfloor \in \mathcal{C}_1.$$

Assume that $\boldsymbol{c} \in \mathcal{C}_2$ and $\boldsymbol{y} = \boldsymbol{c} + \boldsymbol{e}$ is the received word, where $\boldsymbol{e}$ is a t-asymmetric (ℓ, s)-multiple-spaced limited-magnitude error. We use the decoding procedure of $\mathcal{C}_1$, where the input is $\lfloor \frac{1}{s} \cdot \boldsymbol{y} \rfloor$. Note that,

$$\left\lfloor \frac{1}{s} \cdot \boldsymbol{y} \right\rfloor = \left\lfloor \frac{1}{s} \cdot (\boldsymbol{c} + \boldsymbol{e}) \right\rfloor = \left\lfloor \frac{1}{s} \cdot \boldsymbol{c} \right\rfloor + \frac{1}{s} \cdot \boldsymbol{e}.$$

Since $\left\lfloor \frac{1}{s} \cdot \boldsymbol{c} \right\rfloor \in \mathcal{C}_1$, we can consider $\left\lfloor \frac{1}{s} \cdot \boldsymbol{c} \right\rfloor + \frac{1}{s} \cdot \boldsymbol{e}$ to be the input to the decoder of $\mathcal{C}_1$, where $\frac{1}{s} \cdot \boldsymbol{e}$ is a t-asymmetric ℓ-limited-magnitude error. Thus, the decoder of $\mathcal{C}_1$ can decode the error vector $\frac{1}{s} \cdot \boldsymbol{e}$ and multiplying it by s gives with the original error vector $\boldsymbol{e}$. ■

A code will be called *perfect* if it attains the sphere packing bound for t-asymmetric ℓ-limited-magnitude errors [5].

Theorem 5.3.5. *If the code $\mathcal{C}_1$ is perfect and $s|q$, then the code $\mathcal{C}_2$ is perfect as well.*

Proof. If the code $\mathcal{C}_1$ is perfect then

$$|\mathcal{C}_1| \cdot \sum_{i=0}^{t} \binom{n}{i} \ell^i = (q')^n,$$

where $q' = \frac{q}{s}$. The size of the code $\mathcal{C}_2$ is $|\mathcal{C}_2| = s^n \cdot |\mathcal{C}_1|$ and the number of errors is $\sum_{i=0}^{t} \binom{n}{i} \ell^i$. Therefore,

$$|\mathcal{C}_2| \cdot \sum_{i=0}^{t} \binom{n}{i} \ell^i = s^n \cdot |\mathcal{C}_1| \cdot \sum_{i=0}^{t} \binom{n}{i} \ell^i = s^n \cdot (q')^n = q^n$$

and the code $\mathcal{C}_2$ is perfect as well. ■

Theorem 5.3.4 gives us the construction as well as the decoding procedure for the new code $\mathcal{C}_2$. Its encoding procedure is derived from the encoding procedure of $\mathcal{C}_1$. Assume that $\mathcal{C}_1$ is constructed as described earlier in this section using a base code Σ of length n and redundancy r which corrects t symbol errors over an alphabet of size $\ell + 1$ and it has a systematic encoder. Then, Σ is also the base code for $\mathcal{C}_2$. For the simplicity of the encoder we assume that $s(\ell+1)|q$. The encoder's input is a vector

$$(\boldsymbol{u}_1, \boldsymbol{u}_2) \in \{0, \ldots, q-1\}^{n-r} \times \left\{0, \ldots, \frac{q}{\ell+1} - 1\right\}^r.$$

Let $\boldsymbol{v}_2 \in \{0, \ldots, \ell\}^n$ be the encoder's output of Σ when applied to $\left(\left\lfloor \frac{\boldsymbol{u}_1}{s} \right\rfloor \bmod (\ell+1)\right)$. The encoder's output of $\mathcal{C}_2$ is $\boldsymbol{c} = (\boldsymbol{c}_1, \boldsymbol{c}_2)$, where $\boldsymbol{c}_1 = \boldsymbol{u}_1$ and

$$\boldsymbol{c}_2 = s \cdot \left((\ell+1) \left\lfloor \frac{\boldsymbol{u}_2}{s} \right\rfloor + \boldsymbol{v}_2 \right) + (\boldsymbol{u}_2 \bmod s).$$

The vector $\boldsymbol{c}$ satisfies $\boldsymbol{c} \in \{0, \ldots, q-1\}^n$, $\left\lfloor \frac{1}{s} \cdot \boldsymbol{c} \right\rfloor \in \mathcal{C}_1$ and the outputs of two different input vectors are different. Thus, the encoding procedure follows the construction of $\mathcal{C}_2$.

Remark 5.3.2. Let us explain the intuition behind this construction. Assume that q is a power of two and every cell level is represented as a sequence of $\log_2 q$ bits. If we construct asymmetric error-correcting codes where $\ell = 1$, then the base code Σ is binary and the encoding and decoding of the q-ary code are implemented on the LSB of each cell. For asymmetric (ℓ, s)-multiple-spaced limited-magnitude error-correcting codes, assume that $\ell = 2$ and s is also a power of two, say the i-th power, where $2 \leqslant i < \log_2 q - 1$, then the base code Σ is again binary and the encoding and decoding of the q-ary code are implemented on the i-th digit of each cell.

5.4 A Construction of (t_1, t_2)-Asymmetric (ℓ_1, ℓ_2)-Limited-Magnitude Error-Correcting Codes

In this section, we give a construction of (t_1, t_2)-asymmetric (ℓ_1, ℓ_2)-limited-magnitude error-correcting codes. The construction uses the codes proposed by Cassuto et al. [5] which were reviewed in Section 5.3. We will describe the encoding procedure and then show its correctness by the success of its decoding procedure.

Construction 5.4.1. Let t_1, t_2, ℓ_1, ℓ_2 be positive integers such that $\ell_1 < \ell_2$, and let $\ell_2' = \left\lfloor \frac{\ell_2}{\ell_1+1} \right\rfloor$. Let $\mathcal{C}_1$ be an $[n, q, t_1 + t_2, \ell_1]$ error-correcting code and let $\mathcal{C}_2$ be an $[n, q, t_2, \ell_2', \ell_1 + 1]$ error-correcting code. Let Σ_1 and Σ_2 be the base codes that are used to generate the codes $\mathcal{C}_1$ and $\mathcal{C}_2$, respectively. Both base codes are of length n, and they have redundancy r_1 and r_2, respectively. They also have systematic encoders. We construct the code $\mathcal{C}$ by means of the following encoding procedure. The input to the encoder is a vector

$$(\boldsymbol{u}_1, \boldsymbol{u}_2, \boldsymbol{u}_3) \in \{0, \ldots, q-1\}^{n-r_1-r_2} \times \left\{0, \ldots, \frac{q}{\ell_2'+1} - 1\right\}^{r_2} \times \left\{0, \ldots, \frac{q}{\ell_1+1} - 1\right\}^{r_1}.$$

The encoding of these information symbols is carried out in two steps. First, let $\boldsymbol{v}_2$ be the systematic encoder's output of Σ_2 applied to the vector

$$\left(\left\lfloor \frac{\boldsymbol{u}_1}{\ell_1+1} \right\rfloor \bmod (\ell_2'+1), \left\lfloor \frac{\boldsymbol{u}_3}{\ell_1+1} \right\rfloor \bmod (\ell_2'+1) \right),$$

and let

$$\boldsymbol{u}_2' = (\ell_1+1)\left((\ell_2'+1) \left\lfloor \frac{\boldsymbol{u}_2}{\ell_1+1} \right\rfloor + \boldsymbol{v}_2 \right) + (\boldsymbol{u}_2 \bmod (\ell_1+1)).$$

Then, we calculate $\boldsymbol{v}_3$ to be the systematic encoder's output of Σ_1 applied to $(\boldsymbol{u}_1 \bmod (\ell_1 + 1), \boldsymbol{u}_2' \bmod (\ell_1+1))$. Finally, the encoder's output is $\boldsymbol{c} = (\boldsymbol{c}_1, \boldsymbol{c}_2, \boldsymbol{c}_3)$, where $\boldsymbol{c}_1 = \boldsymbol{u}_1, \boldsymbol{c}_2 = \boldsymbol{u}_2'$ and $\boldsymbol{c}_3 = (\ell_1+1) \cdot \boldsymbol{u}_3 + \boldsymbol{v}_3$.

Remark 5.4.1. We assume here that $r_1 + r_2 \leqslant n$. However, if this is not the case we can modify the construction to be applicable in this scenario.

Before we show the correctness of this construction, let us prove a few properties of (t_1, t_2)-asymmetric (ℓ_1, ℓ_2)-limited-magnitude errors. Assume that $\boldsymbol{e}$ is a (t_1, t_2) asymmetric (ℓ_1, ℓ_2)-limited-magnitude error. First note that this error vector can be written as $\boldsymbol{e} = \boldsymbol{e}_1 + \boldsymbol{e}_2$, where

$$\boldsymbol{e}_1 = \boldsymbol{e} \bmod (\ell_1 + 1), \;\; \boldsymbol{e}_2 = \boldsymbol{e} - \boldsymbol{e}_1.$$

Lemma 5.4.1. *For all* $\boldsymbol{c} \in \mathcal{C}_1$, $\boldsymbol{c} + \boldsymbol{e}_2 \in \mathcal{C}_1$.

Proof. The proof follows from the observation that

$$\boldsymbol{e}_2 \bmod (\ell_1 + 1) = \mathbf{0}.$$

■

Lemma 5.4.2. *The error vector* $\boldsymbol{e}_1$ *is a* $(t_1 + t_2)$*-asymmetric* ℓ_1*-limited-magnitude error.*

Proof. Since $wt(\boldsymbol{e}) \leqslant t_1 + t_2$ so is $wt(\boldsymbol{e}_1) \leqslant t_1 + t_2$ and clearly $\max_{0 \leqslant i \leqslant n-1}\{e_{1,i}\} \leqslant \ell_1$. ■

Lemma 5.4.3. *The vector* $\boldsymbol{e}_2$ *is a* t_2*-asymmetric* $(\ell_2', \ell_1 + 1)$*-multiple-spaced limited-magnitude error.*

Proof. For each i, $1 \leqslant i \leqslant n$, if $e_i \leqslant \ell_1$ then $e_{2,i} = 0$ and therefore $wt(\boldsymbol{e}_2) \leqslant t_2$. Furthermore,

$$e_{2,i} = e_i - e_{1,i} = e_i - (e_i \bmod (\ell_1 + 1)),$$

and thus $e_{2,i} \equiv 0 \bmod (\ell + 1)$. Finally,

$$\begin{aligned} e_{2,i} &= e_i - (e_i \bmod (\ell_1 + 1)) = \left\lfloor \frac{e_i}{\ell_1 + 1} \right\rfloor \cdot (\ell_1 + 1) \\ &\leqslant \left\lfloor \frac{\ell_2}{\ell_1 + 1} \right\rfloor \cdot (\ell_1 + 1) = \ell_2' \cdot (\ell_1 + 1) \end{aligned}$$

■

In the next theorem we will prove the correctness of this construction by showing the success of its decoding procedure.

Theorem 5.4.4. *The code $\mathcal{C}$ generated by Construction 5.4.1 is an $[n, q, (t_1, t_2), (\ell_1, \ell_2)]$ error-correcting code.*

Proof. The proof follows from the decoding procedure of the code $\mathcal{C}$. Assume the received word is $\boldsymbol{y} = \boldsymbol{c} + \boldsymbol{e}$ where $\boldsymbol{e}$ is a (t_1, t_2)-asymmetric (ℓ_1, ℓ_2)-limited-magnitude error vector. As mentioned above, we write the error vector in the form $\boldsymbol{e} = \boldsymbol{e}_1 + \boldsymbol{e}_2$, where $\boldsymbol{e}_1 = \boldsymbol{e} \bmod (\ell_1 + 1)$ and $\boldsymbol{e}_2 = \boldsymbol{e} - \boldsymbol{e}_1$. From Lemma 5.4.1, $\boldsymbol{c} + \boldsymbol{e}_2 \in \mathcal{C}_1$ and from Lemma 5.4.2, $\boldsymbol{e}_1$ is a $(t_1 + t_2)$-asymmetric ℓ_1-limited magnitude error vector. Therefore, by applying the decoder of $\mathcal{C}_1$ to the received word $\boldsymbol{y}$, the error vector $\boldsymbol{e}_1$ is decoded and the decoder's output is $\boldsymbol{y}' = \boldsymbol{c} + \boldsymbol{e}_2$.

According to Lemma 5.4.3, the error vector $\boldsymbol{e}_2$ is a t_2-asymmetric $(\ell_2', \ell_1 + 1)$-multiple-spaced limited-magnitude error. However, note that $\boldsymbol{c}$ is not a codeword of $\mathcal{C}_2$. In fact, $\boldsymbol{c}$ is the encoded codeword, which is the output of the encoder of $\mathcal{C}_1$. Its input, which is the output of the encoder of $\mathcal{C}_2$, is $\boldsymbol{c}'$, where for all $1 \leqslant i \leqslant n - r_1$, $c_i' = c_i$, and for $n - r_1 + 1 \leqslant i \leqslant n$, $c_i' = c_i - (c_i \bmod (\ell_1 + 1))$. But, the decoder of $\mathcal{C}_2$ decodes a t_2-asymmetric $(\ell + 1)$-multiple ℓ_2'-limited-magnitude error by calculating

$$\left\lfloor \frac{1}{\ell_1 + 1} \cdot \boldsymbol{y}' \right\rfloor = \left\lfloor \frac{1}{\ell_1 + 1} \cdot (\boldsymbol{c} + \boldsymbol{e}_2) \right\rfloor$$
$$= \left\lfloor \frac{1}{\ell_1 + 1} \cdot \boldsymbol{c} \right\rfloor + \frac{1}{\ell_1 + 1} \cdot \boldsymbol{e}_2 = \left\lfloor \frac{1}{\ell_1 + 1} \cdot \boldsymbol{c}' \right\rfloor + \frac{1}{\ell_1 + 1} \cdot \boldsymbol{e}_2.$$

Therefore, the error output of the decoder of $\mathcal{C}_2$ is $\boldsymbol{e}_2$ and we successfully receive the transmitted codeword $\boldsymbol{c}$. ■

In order to evaluate this code construction we compare it to the codes by Cassuto el al. [5]. Clearly, for all positive integers t_1, t_2, ℓ_1, ℓ_2 such that $\ell_1 \leqslant \ell_2$, every $[n, q, t_1 + t_2, \ell_2]$ error-correcting code is also an $[n, q, (t_1, t_2), (\ell_1, \ell_2)]$ error-correcting code. In case that t_1 and t_2 are roughly the same, it turns out that our construction is inferior. The reason is that the number of errors found by $\mathcal{C}_1$ is $t_1 + t_2$ and the number of errors found by $\mathcal{C}_2$ is t_2. Even though the magnitude of the errors is smaller than ℓ_2, the total number of errors found by the two codes is $t_1 + 2t_2$, as opposed to $t_1 + t_2$ errors corrected by an $[n, q, t_1 + t_2, \ell_2]$ code. Since the sizes of the two codes depend on the sizes of their base codes, in order to give an accurate comparison, one needs to know the exact sizes of these base codes. If all the base codes were perfect or close to be perfect then it is possible to show that, approximately, if $\frac{t_1}{t_2} > \frac{\log n}{\log \ell_2 - \log \ell_1}$, then our scheme is superior. For example, if $n = 1000$ and $\ell_1 = 1, \ell_2 = 4, t_1 = 6, t_2 = 1$, our construction yields better codes. Consider another example of $[n, q, (n - 1, 1), (1, 2)]$ error-correcting codes, where $12|q$. Then, the size of the best $[n,q,n,2]$ error-correcting codes will be $\left(\frac{q}{3}\right)^n$, while our construction achieves codes of size $\frac{1}{2^{\lceil \log n \rceil}} \cdot \left(\frac{q}{2}\right)^n > \left(\frac{q}{3}\right)^n$.

5.5 A Construction of $(1,1)$-Asymmetric $(1,\ell)$-Limited-Magnitude Error-Correcting Codes

We saw in the previous section that if the values of t_1 and t_2 are roughly the same then our construction does not necessarily outperform the construction by Cassuto et al. Here, we consider one case where it is possible to achieve better code constructions. We start with a construction of an $[n,q,(1,1),(1,2)]$ error-correcting code.

Theorem 5.5.1. *Let q,m be positive integers such that $m > 1$ and $3|q$, and $\mathcal{C}_1$ is the code constructed in Theorem 5.3.2 where $n = 3^m - 1, p = 3, t = 2$. Then, the code $\mathcal{C}$, defined as,*

$$\mathcal{C} = \{\boldsymbol{c} \in \{0,\ldots,q-1\}^n | \boldsymbol{c}(\bmod 3) \in \mathcal{C}_1, \sum_{i=1}^{n} c_i \equiv 0(\bmod 2)\}.$$

is an $[n,q,(1,1),(1,2)]$ error-correcting code.

Proof. Let $\boldsymbol{c}$ be the transmitted codeword, $\boldsymbol{y} = \boldsymbol{c} + \boldsymbol{e}$ the received word where $\boldsymbol{e}$ is a $(1,1)$-asymmetric $(1,2)$-limited magnitude error, and $s_1 = \sum_{i=1}^{n} y_i\alpha^i, s_2 = \sum_{i=1}^{n} y_i\alpha^{2i}$. The sum of the received symbols can have either odd or even parity, modulo 2.
Odd sum-parity: $\sum_{i=1}^{n} y_i \equiv 1(\bmod 2)$.

There are two possible cases.

1. The weight of $\boldsymbol{e}$ is one and the error magnitude is one.
2. The weight of $\boldsymbol{e}$ is two, one error is of magnitude one and the other one is of magnitude two.

In the first case, we get $s_1 = \alpha^i, s_2 = \alpha^{2i} = s_1^2$, where i is the error location. In the second case, $s_1 = \alpha^{i_1} + 2\alpha^{i_2}, s_2 = \alpha^{2i_1} + 2\alpha^{2i_2}$, where i_1, i_2 are the error locations and

$$\begin{aligned} s_1^2 &= (\alpha^{i_1} + 2\alpha^{i_2})^2 = \alpha^{2i_1} + \alpha^{2i_2} + \alpha^{i_1+i_2} \\ &= \alpha^{2i_1} + 2\alpha^{2i_2} - (\alpha^{2i_2} - \alpha^{i_1+i_2}) = s_2 + \alpha^{i_2}(\alpha^{i_2} - \alpha^{i_1}) \neq s_2. \end{aligned}$$

Hence we can distinguish between these two cases. The error location error in the first case is easy to find. In the second case, we decode as follows:

$$\begin{aligned}\frac{s_2}{s_1}+s_1&=\frac{\alpha^{2i_1}+2\alpha^{2i_2}}{\alpha^{i_1}+2\alpha^{i_2}}+\alpha^{i_1}+2\alpha^{i_2}\\ &=\frac{\alpha^{2i_1}-\alpha^{2i_2}}{\alpha^{i_1}-\alpha^{i_2}}+\alpha^{i_1}-\alpha^{i_2}=2\alpha^{i_1}.\end{aligned}$$

From this we can determine the location of the error with magnitude one and, therefore, also the location of the error with magnitude two.

Even sum-parity: $\sum_{i=1}^{n} y_i \equiv 0 (\bmod 2)$. There are three cases:

1. There is no error.
2. The weight of $\boldsymbol{e}$ is one and the error magnitude is two.
3. The weight of $\boldsymbol{e}$ is two and both errors have magnitude one.

In the first case, we get $s_1 = s_2 = 0$. In the second case, $s_1 = 2\alpha^i, s_2 = 2\alpha^{2i} = 2s_1^2$, where i is the error location. In the third case, we can show as before that $s_2 \neq 2s_1^2$. Hence, we can distinguish between the three cases and the error locations in each case can again be easily determined. ■

Assume q is even. The size of the code $\mathcal{C}_1$ is 3^{n-2m} and, as defined, the size of $\mathcal{C}$ is $\frac{q^n}{2\cdot 9^m}$. On the other hand, suppose that we use Construction 5.3.1 to design an $[n, q, 2, 2]$ error correcting code $\mathcal{C}'$ with the same parameters n and q. If the base code Σ is an optimal linear code that corrects two errors over $GF(3)$, its redundancy is at least $\lceil \log_3(2n^2+1) \rceil = 2m+1$, and therefore the size of the code $\mathcal{C}'$ is at most $\frac{q^n}{3^{2m+1}} < \frac{q^n}{2\cdot 9^m}$.

5.6 Conclusion

In this work, we studied a new error model for multi-level flash memories based upon a graded distribution of asymmetric errors of limited magnitudes. Using a recent construction by Cassuto et al. [5] of asymmetric limited-magnitude error-correcting codes, we developed a family of codes that correct asymmetric errors with magnitudes a multiple of some fixed integer. We then utilized these two classes of codes to construct codes that correct t_1 asymmetric errors of magnitude no more than ℓ_1 and t_2 errors of magnitude no more than ℓ_2, where $\ell_1 < \ell_2$. Finally, we discussed efficient constructions for the special case where $t_1 = t_2 = 1$, $\ell_1 = 1$, and $\ell_2 > 1$.

Acknowledgment

This chapter is in part a reprint of the material in the paper: E. Yaakobi, P.H. Siegel, A. Vardy, and J.K. Wolf, "On Codes that Correct Asymmetric Errors with Graded Multiple Distribution," to appear *IEEE International Symposium on Information Theory*, Saint Petersburg, Russia, July 2011.

Bibliography

[1] R. Ahlswede, H. Aydinian, L. Khachatrian, and L. Tolhuizen, "On q-ary codes correcting all unidirectional errors of a limited magnitude," Arxiv.org, http://arxiv.org/abs/cs/0607132, Tech. Rep. cs.IT/0607132, 2006.

[2] S. Al-Bassam, R. Venkatesan, and S. Al-Muhammadi, "New single asymmetric error-correcting codes," *IEEE Trans. Inform. Theory*, vol. 43, no. 5, pp. 1619–1623, September 1997.

[3] J.M. Berger, "A note on error detecting codes for asymmetric channels," *Inform. and Contr.*, vol. 4, pp. 68–73, March 1961.

[4] B. Bose and S. Al-Bassam, "On systematic single asymmetric error-correcting codes," *IEEE Trans. Inform. Theory*, vol. 46, no. 2, pp. 669–772, March 2000.

[5] Y. Cassuto, M. Schwartz, V. Bohossian, and J. Bruck, "Codes for asymmetric limited-magnitude errors with application to multi-level flash memories," *IEEE Trans. Inform. Theory*, vol. 56, no. 4, pp. 1582–1595, April 2010.

[6] C.M. Compagnoni, A.S. Spinelli, R. Gusmeroli, A.L. Lacaita, S. Beltrami, A. Ghetti and A. Visconti, "First evidence for injection statistics accuracy limitations in NAND Flash constant-current Fowler-Nordheim programming," *Proc. IEEE Int. Electron Devices Meeting*, pp. 165–168, December 2007.

[7] S.D. Constantin and T.R.N. Rao, "On the theory of binary asymmetric error-correcting codes," *Inform. and Contr.*, vol. 40, pp. 20-36, January 1979.

[8] P. Delsarte and P. Piret, "Bounds and constructions for binary asymmetric error-correcting codes," *IEEE Trans. Inform. Theory*, vol. 27, no. 1, pp. 125–128, January 1981.

[9] L. Dolececk, "Towards longer lifetime of emerging memory technologies using number theory," *Proc. Workshop on the Application of Communication Theory to Emerging Memory Technologies*, Miami, Florida, December 2010.

[10] N. Elarief and B. Bose, "Optimal, systematic, q-ary codes correcting all asymmetric and symmetric errors of limited magnitude," *IEEE Trans. Inform. Theory*, vol. 56, no. 3, pp. 979–983, March 2010.

[11] T. Etzion, "New lower bounds for asymmetric and unidirectional codes," *IEEE Trans. Inform. Theory*, vol. 37, no. 6, pp. 1696–1705, November 1991.

[12] F. Fu, A. Ling, and C. Xing, “New lower bounds and constructions for binary codes correcting asymmetric errors,” *IEEE Trans. Inform. Theory*, vol. 49, no. 12, pp. 3294–3299, December 2003.

[13] C.V. Freeman, “Optimal error detection codes for completely asymmetric binary channels,” *Inform. and Contr.*, vol. 5, no. 1, pp. 64–71, March 1962.

[14] T. Helleseth and T. Kløve, “On group-theoretic codes for asymmetric channels,” *Inform. and Contr.*, vol. 49, pp. 1–9, 1981.

[15] T. Kløve, “Error correcting codes for the asymmetric channel,” Dept. Mathematics, University of Bergen, Norway, Tech. Rep. 18-09-07-81, 1981.

[16] T. Kløve, B. Bose, and N. Elarief, “Systematic single limited magnitude asymmetric error correcting codes,” *Proc. IEEE Inform. Theory Workshop*, Cairo, Egypt, January 2010.

[17] N. Mielke, T. Marquart, N. Wu, J. Kessenich, H. Belgal, E. Schares, F. Triverdi, E. Goodness, L.R. Nevill, “Bit error rate in NAND flash memories,” *Proceedings of IEEE Reliability Physics Symposium*, pp. 9–19, May 2008.

[18] R. Seroul and D. O’Shea, *Programming for Matematicians*, Springer, 2000.

[19] R.R. Varshamov, “On some specifics of asymmetric error-correcting linear codes,” *Rep. Acad. Sci. USSR*, vol. 157, no. 3, pp. 546–548, 1964.

[20] R.R. Varshamov and G.M. Tenenholtz, “A code for correcting a single asymmetric error,” *Automatica i Telemekhanika*, vol. 26, no. 2, pp. 288–292, 1965.

[21] J.H. Weber, C. de Vroedt, and D.E. Boekee, “Bounds and constructions for binary codes of length less than 24 and asymmetric distance less than 6,” *IEEE Trans. Inform. Theory*, vol. 34, no. 5, pp. 1321–1331, September 1988.

[22] E. Yaakobi, J. Ma, L. Grupp, P.H. Siegel, S. Swanson, and J.K. Wolf, “Error characterization and coding schemes for flash memories,” *Proc. Workshop on the Application of Communication Theory to Emerging Memory Technologies*, Miami, Florida, December 2010.

Chapter 6

On The Parallel Programming of Flash Memory Cells

6.1 Introduction

Parallel programming is an important tool used in flash memories to achieve high write speed. Studying how cells are programmed is crucial for understanding the storage capacity of flash memories. While programming the cells, charge can be progressively injected into a cell; however, in order to remove charge from any cell, a large block of cells (about 10^6 cells) must be first erased together (i.e., all charge in them be fully removed) before reprogrammed [3]. Therefore, in order to avoid block erasures, flash memory cells are usually programmed cautiously with multiple rounds of charge injection, so that every cell's level gradually approaches its target level [13, 18].

Parallel programming in flash memories has two important properties: *shared program voltages*, and *variation in charge-injection properties*. A flash memory injects charge into a cell by applying a program voltage to the cell. In parallel programming, a common program voltage is applied to many cells for simultaneous charge injection [3]. Compared to applying a separate program voltage to each cell, this property significantly simplifies the complexity of the memory hardware. It is also a constraint for the storage capacity of flash memories. Another important property is that cells have different hardness for charge injection [13, 18]. Some cells are *easy* to program, while others are *hard* to program. When the same program voltage is applied to cells, the easier-to-program cells will have more charge injected into them than the harder-to-program cells. This hardness is an intrinsic property of each cell. To accurately control the final cell levels, different program voltages should be applied to cells based on their hardness for programming. A widely used method in industry is the *Incremental Step Pulse Programming* (ISPP) scheme [13,18], which gradually increases the program voltage to first program the easier cells and then the harder cells. To ensure that the (easier) cells are not overprogrammed, the subsequent program voltages are not applied to the cells whose levels are already sufficiently close to their target levels.

In previous works, the optimal programming of single flash memory cells has been studied. In [7], an optimal programming algorithm was presented for storing as much data as possible in a single cell, which achieves the zero-error storage capacity under its programming noise model. In [8], an algorithm was shown for optimizing the expected cell programming precision, when the programming noise follows a random distribution. Note that for programming a single cell, the two important properties for parallel programming – *shared program voltages* and *variation in programming hardness* – are not considered. So for parallel programming, new techniques need to be developed.

6.2 Terms and Concepts

Let us first describe the general information theoretic framework of the cell programming problem. This problem can be modeled as a cascade channel, described in Figure 7.1, where the number of channels is as the number of programming rounds, t. We assume that there are n cells, whose initial *levels* (i.e., charge levels) are all 0. When the cells are programmed, a cell's level can only increase. Each cell is characterized by some target level $\theta_i \geqslant 0$ and the target levels vector is $\Theta \triangleq (\theta_1, \ldots, \theta_n)$. Each round of programming is first described by an encoder E_i, $1 \leqslant i \leqslant t$. The input to the first encoder is the target levels vector and for the other encoders the input includes also a feedback on the cells' level based on the previous programming round. The output of each encoder is the variable X_i which includes an information about the program voltage on the i-th round and the set of cells that are applied by this voltage. Then, the output of the channel, which is the outcome of the programming iteration on the i-th round, is a function of X_i, N_i, and Y_{i-1} if $i > 1$. The variable N_i represents the noise in each cell and any other property of the cell that will affect its level. The variable Y_{i-1} represents the current value of each cell. For the next round, the outcome on the i-th round of programming Y_i is used in order to generate a feedback F_i on the cell levels which will be used in the next programming round. Then, the goal is to minimize a cost function between the channel output Y_t after t programming rounds and the target levels vector Θ.

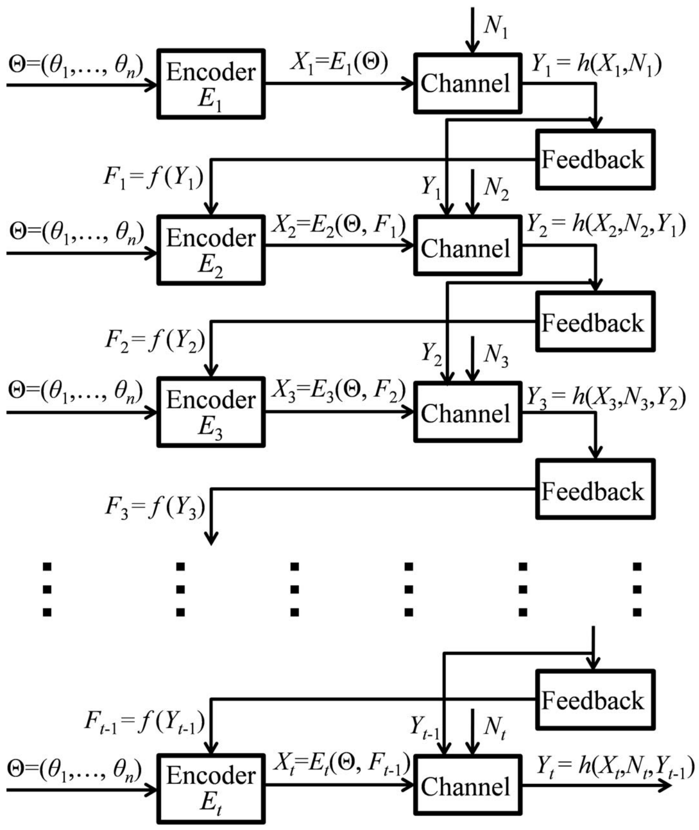


Figure 6.1: The information theoretic framework of the cell programming model.

There are different ways to specify this general framework. For example, this model of programming the cells can be also seen as a rewritable channel, which was described by Lastras-Montaño et al. [11,12,15]. Yet another model was given in [17] which approximates the cells' levels after programming while taking into account other effects like inter-cell interference and aging of the flash memory blocks. In this work, we focus on the following model. Let $c_1, c_2, \ldots, c_n$ be the n flash memory cells. We model the hardness of charge injection for the cells by the positive parameters $\alpha_1, \alpha_2, \ldots, \alpha_n$. Specifically, for $i \in [n] \triangleq \{1, 2, \ldots, n\}$, if a program voltage V is used to inject charge into the cell c_i, the level of c_i will increase by

$$\alpha_i V + \epsilon,$$

where ϵ is called the *programming noise* and is a random number with a probabilistic distribution. (The distribution of ϵ may depend on α_i and V.) In this work, we will mostly consider the case $\epsilon = 0$ (which means the programming noise is sufficiently small), and focus on studying how α_i (i.e., the variance in charge-injection hardness) impacts parallel programming.

For $i \in [n]$, $\theta_i \geqslant 0$ is the target level of the cell c_i. The programming process consists of t rounds of charge injection, and the goal is to make the final level of c_i be very close to θ_i. To guarantee high write speed, t is usually a small constant (e.g., $t = 6$ or 8 for MLC flash). Let $V_1, V_2, \ldots, V_t$ denote the program voltages of the t rounds of programming, respectively. In each round, the memory can decide whether to apply the program voltage to a cell or not. For $i \in [n]$ and $j \in [t]$, let $b_{i,j}$ be a 0/1 integer such that in the j-th round of programming, if the program voltage V_j is applied to the cell c_i, then $b_{i,j} = 1$; otherwise, $b_{i,j} = 0$. For $i \in [n]$ and $j \in [t]$, let $\ell_{i,j}$ denote the level of c_i after the j-th round of programming. Then we have

$$\ell_{i,j} = \alpha_i \cdot \left(V_1, V_2, \cdots, V_j\right) \cdot \left(b_{i,1}, b_{i,2}, \cdots, b_{i,j}\right)^T.$$

We measure the performance of programming by a cost function $\mathcal{C}(\Theta, L)$, where $\Theta = (\theta_1, \theta_2, \ldots, \theta_n)$ are the target levels and $L \triangleq (\ell_{1,t}, \ell_{2,t}, \ldots, \ell_{n,t})$ are the final cell levels. There are various ways to define $\mathcal{C}(\Theta, L)$. We will adopt the ℓ_p metric and denote it by $\mathcal{C}_p(\Theta, L)$:

$$\mathcal{C}_p(\Theta, L) \triangleq \left(\sum_{i=1}^{n} |\theta_i - \ell_{i,t}|^p\right)^{1/p}.$$

Given the integer p, our objective is to minimize $\mathcal{C}_p(\Theta, L)$.

In the parallel programming problem, Θ and t are given parameters. We can choose $V_1, \ldots, V_t$ and $b_{1,1}, \ldots, b_{n,t}$ for the best performance. It makes a difference whether we know the values of $\alpha_1, \ldots, \alpha_n$, and whether we can learn the cell levels after each programming round (which is feedback information). If the feedback information on cell levels is available, the programming algorithm can be adaptive. Note that it is very time consuming to measure the exact cell levels. In practice, it is much faster to compare the cell levels with some preset threshold levels (using comparators), and use the information on cell levels to make decisions on programming [13, 18]. We show examples in the following.

Example 6.2.1. Let $n = 8$,

$$\Theta = (1.0, 1.0, 2.0, 2.0, 1.0, 2.0, 2.0, 2.0),$$

$t = 2$, and

$$(\alpha_1, \ldots, \alpha_8) = (0.5, 0.5, 0.8, 0.75, 0.5, 0.42, 0.85, 0.46),$$

be known parameters. Assume there is no feedback information on the cell levels after each round of programming.

Suppose we choose $V_1 = 2.0$ and $V_2 = 2.5$, and choose

$$\begin{pmatrix} b_{1,1} & b_{2,1} & \cdots & b_{8,1} \\ b_{1,2} & b_{2,2} & \cdots & b_{8,2} \end{pmatrix} = \begin{pmatrix} 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 \end{pmatrix}.$$

Then we get

$$L = (1.0, 1.0, 2.0, 1.875, 1.0, 1.89, 2.125, 2.07),$$

and

$$\mathcal{C}_p(\Theta, L) = \left(2 \times 0.125^p + 0.11^p + 0.07^p\right)^{1/p}.$$

If $p = 2$, then the programming performance is $\mathcal{C}_2(\Theta, L) = 0.220$.

Example 6.2.2. Let $n \geqslant 1$, $\Theta = (1.0, 1.0, \ldots, 1.0)$, and $t = 2$. Let $\alpha_{\min}$ and $\alpha_{\max}$ be two known parameters such that $0 < \alpha_{\min} < \alpha_{\max}$. Suppose that the values of $\alpha_1, \cdots, \alpha_n$ are unknown; however, it is known that $\alpha_i \in [\alpha_{\min}, \alpha_{\max}]$ for $i \in [n]$. Suppose that we have feedback information on cell levels in the following way: after the first round of programming, we can compare all cells with a common preset threshold level τ to see if their levels are above or below τ. Based on this information, the memory can adaptively adjust the second round of programming. Suppose that we want to minimize $\mathcal{C}_\infty(\Theta, L)$.

It can be verified that the following programming algorithm is optimal. Choose

$$V_1 = \frac{2}{\sqrt{\alpha_{\max}}\left(\sqrt{\alpha_{\max}} + \sqrt{\alpha_{\min}}\right)},$$
$$V_2 = \frac{2\left(\sqrt{\alpha_{\max}} - \sqrt{\alpha_{\min}}\right)}{\sqrt{\alpha_{\max}\alpha_{\min}}\left(\sqrt{\alpha_{\max}} + \sqrt{\alpha_{\min}}\right)},$$
$$\tau = \frac{2\sqrt{\alpha_{\min}}}{\sqrt{\alpha_{\max}} + \sqrt{\alpha_{\min}}}.$$

After the first round of programming, all the cell levels are in the range

$$[\alpha_{\min}V_1, \alpha_{\max}V_1] = \left[\frac{2\alpha_{\min}}{\sqrt{\alpha_{\max}}\left(\sqrt{\alpha_{\max}} + \sqrt{\alpha_{\min}}\right)}, \frac{2\sqrt{\alpha_{\max}}}{\sqrt{\alpha_{\max}} + \sqrt{\alpha_{\min}}}\right]$$

(Note that $\tau < 1$, $\alpha_{\max}V_1 > 1$, and $1 - \tau = \alpha_{\max}V_1 - 1$.)

Let $\mathcal{S}$ denote the set of cells whose levels are less than τ after the first round of programming. (Note that for any cell $c_i \in \mathcal{S}$, we know now that $\alpha_i \in [\alpha_{\min}, \frac{\tau}{V_1})$.) Only the cells in S are programmed in the second round. After the second round, the levels of the cells in $\mathcal{S}$ are in the range

$$[\alpha_{\min}(V_1 + V_2), \frac{\tau}{V_1} \cdot (V_1 + V_2)) = [\tau, \alpha_{\max}V_1).$$

So are the cells in $\{c_1, \ldots, c_n\} \setminus \mathcal{S}$. So we get

$$\mathcal{C}_\infty(\Theta, L) = 1 - \tau = \alpha_{\max}V_1 - 1 = \frac{\sqrt{\alpha_{\max}} - \sqrt{\alpha_{\min}}}{\sqrt{\alpha_{\max}} + \sqrt{\alpha_{\min}}}.$$

6.3 Optimal Programming without Feedback

In this section, we present an optimal programming algorithm when the cells' hardness for programming – $\alpha_1, \alpha_2, \ldots, \alpha_n$ – is known. In this case, there is no need to obtain the feedback information on cell levels during programming, because they change deterministically. (In practice, $\alpha_1, \ldots, \alpha_n$ may be estimated values through some cell profiling process.) Let n, t, $\theta_1, \ldots, \theta_n$ and p also be known parameters. The programming algorithm needs to find $V_1, \ldots, V_t$ and $b_{1,1}, \ldots, b_{n,t}$ that minimize $\mathcal{C}_p(\Theta, L)$.

The parallel programming problem here is similar, for $p = 2$, to the Subspace/Subset Selection Problem (SSP) [6] and the Sparse Approximate Solution Problem (SAS) [16], because we can see

$$\vec{p}_i \triangleq (b_{1,i}, b_{2,i}, \ldots, b_{n,i})^T$$

for $i = 1, \ldots, t$ as vectors, see $V_1, \ldots, V_t$ as real coefficients, and the objective is to make the linear combination

$$(\vec{p}_1, \ldots, \vec{p}_t) \cdot (V_1, \ldots, V_t)^T$$

be close to a given vector $(\theta_1, \ldots, \theta_n)^T$. Both SSP and SAS problems are NP hard. However, we show here that the parallel programming problem has a polynomial-time solution (for many commonly used ℓ_p metrics, including $p = 1, 2, \infty$), using the condition that here $t = O(1)$ is a constant.

Without loss of generality (WLOG), we assume that

$$\frac{\theta_1}{\alpha_1} \leqslant \frac{\theta_2}{\alpha_2} \leqslant \cdots \leqslant \frac{\theta_n}{\alpha_n}.$$

(It takes time complexity $O(n \lg n)$ to sort the n ratios and label the n cells this way.)

Lemma 6.3.1. *There exists an optimal solution $V_1, \ldots, V_t, b_{1,1}, \ldots, b_{n,t}$ such that for any $1 \leqslant i < j \leqslant n$,*

$$(V_1, \ldots, V_t) \cdot (b_{i,1}, \ldots, b_{i,t})^T \leqslant (V_1, \ldots, V_t) \cdot (b_{j,1}, \ldots, b_{j,t})^T$$

Proof. For $k = 1, 2, \ldots, n$, let us define

$$y_k \triangleq (V_1, \ldots, V_t) \cdot (b_{k,1}, \ldots, b_{k,t})^T.$$

Note that for each $k \in [n]$, $\mathcal{C}_p(\Theta, L)$ is monotonically increasing in $|\theta_k - \ell_{k,t}| = |\theta_k - \alpha_k y_k|$. Consider any optimal solution where $y_i > y_j$. Since $|\theta_i - \ell_{i,t}|$ is minimized, we have

$$\begin{aligned} |\theta_i - \alpha_i y_i| &\leqslant |\theta_i - \alpha_i y_j|, \\ \theta_i^2 - 2\theta_i\alpha_i y_i + \alpha_i^2 y_i^2 &\leqslant \theta_i^2 - 2\theta_i\alpha_i y_j + \alpha_i^2 y_j^2, \\ \alpha_i^2(y_i^2 - y_j^2) &\leqslant 2\theta_i\alpha_i(y_i - y_j), \\ \frac{y_i + y_j}{2} &\leqslant \frac{\theta_i}{\alpha_i}. \end{aligned}$$

Similarly, since $|\theta_j - \ell_{j,t}|$ is minimized, we have $\frac{\theta_j}{\alpha_j} \leqslant \frac{y_i+y_j}{2}$. Since $\frac{\theta_i}{\alpha_i} \leqslant \frac{\theta_j}{\alpha_j}$, we get $\frac{\theta_i}{\alpha_i} = \frac{y_i+y_j}{2} = \frac{\theta_j}{\alpha_j}$. So

$$|\theta_i - \alpha_i y_i| = |\theta_i - \alpha_i y_j|$$

and we can make $(b_{i,1}, \ldots, b_{i,t})$ take the value of $(b_{j,1}, \ldots, b_{j,t})$ and still get an optimal solution. This way we can convert any optimal solution to an optimal solution that has the property shown in the lemma. ■

Let $B_1, B_2, \ldots, B_{2^t}$ denote the 2^t distinct binary vectors of length t, respectively. And let

$$\mathcal{S}_B \triangleq \{B_1, B_2, \ldots, B_{2^t}\}.$$

(Let $B_1, \ldots, B_{2^t}$ be column vectors, e.g., $(0, \ldots, 0)^T$. Clearly, for $i \in [n]$, $(b_{i,1}, \ldots, b_{i,t})^T \in \mathcal{S}_B$.) There are $2^t!$ permutations for the elements in $\mathcal{S}_B$, which we denote by $\pi_1, \pi_2, \ldots, \pi_{2^t!}$. For $i \in [2^t!]$, we denote the permutation $\pi_i(\mathcal{S}_B)$ by

$$\left(B_1^{\pi_i}, B_2^{\pi_i}, \ldots, B_{2^t}^{\pi_i}\right).$$

Define $\vec{V} \triangleq (V_1, V_2, \ldots, V_t)$. Given the program voltages $V_1, \ldots, V_t$, let $\mathcal{I}(\vec{V})$ be the integer in $[2^t!]$ such that

$$\vec{V} \cdot B_1^{\pi_{\mathcal{I}(\vec{V})}} \leqslant \vec{V} \cdot B_2^{\pi_{\mathcal{I}(\vec{V})}} \leqslant \cdots \leqslant \vec{V} \cdot B_{2^t}^{\pi_{\mathcal{I}(\vec{V})}}.$$

(If there is more than one such integer, $\mathcal{I}(\vec{V})$ can be any one of them.) The following result follows from Lemma 6.3.1.

Lemma 6.3.2. *There exists an optimal solution $V_1, \ldots, V_t, b_{1,1}, \ldots, b_{n,t}$ such that we can partition the set $\{1, 2, \ldots, n\}$ into 2^t subsets*

$$\{1, 2, \ldots, k_1\}, \{k_1+1, k_1+2, \ldots, k_2\}, \ldots, \{k_{i-1}+1, \ldots, k_i\}, \ldots, \{k_{2^t-1}+1, \ldots, n\}$$

with this property: $\forall\ i \in [2^t]$ and $j \in \{k_{i-1}+1, k_{i-1}+2, \ldots, k_i\}$, $(b_{j,1}, b_{j,2}, \ldots, b_{j,t})^T = B_i^{\pi_{\mathcal{I}(\vec{V})}}$. (Here we let $0 = k_0 \leqslant k_1 \leqslant k_2 \leqslant \cdots \leqslant k_{2^t} = n$.)

Proof. According to Lemma 6.3.1, the amount of injected charge to the cell is a nondecreasing function. If there are t rounds of programming, then there are 2^t different amounts of charge to inject to each cell. Each such an amount of charge corresponds to a length-t binary vector describing on each rounds the cell was programmed. Therefore, it is possible to divide the n cells into 2^t groups such that in each group the same amount of charge is injected to all the cells. ■

There are $\binom{n+2^t-1}{2^t-1}$ ways to choose the integers $k_1, k_2, \ldots, k_{2^t-1}$. (Note that if $k_{i-1} = k_i$ for some i, then the subset $\{k_{i-1}+1, \ldots, k_i\}$ is actually empty.) For $i = 1, 2, \ldots, \binom{n+2^t-1}{2^t-1}$, let $(k_1(i), k_2(i), \ldots, k_{2^t-1}(i))$ be the integers chosen for $(k_1, k_2, \ldots, k_{2^t-1})$ in the i-th way.

We present the parallel programming algorithm. The idea is to fix each permutation π_i and the partitioning integers $k_1, \ldots, k_{2^t-1}$, and search for the optimal solution.

Algorithm 6.3.3 PROGRAMMING WITHOUT FEEDBACK

```
    Let $f_{opt} \leftarrow \infty$.
    Let $\vec{V}_{opt} \leftarrow (-1, -1, \ldots, -1)$. ($\vec{V}_{opt}$ has length $t$.)
```

```
Let $b_{i,j}^{opt} \leftarrow -1$ for $i \in [n]$ and $j \in [t]$.
For $i = 1, 2, \ldots, 2^t!$
{ For $j = 1, 2, \ldots, \binom{n+2^t-1}{2^t-1}$
        { Find $V_1, V_2, \ldots, V_t$ that minimize the function $f \triangleq$
          $\sum_{d=1}^{2^t} \sum_{q=k_{d-1}(j)+1,k_{d-1}(j)+2,\ldots,k_d(j)} \left|\theta_q - \alpha_q \cdot (V_1, V_2, \ldots, V_t) \cdot B_d^{\pi_i}\right|^p$
              subject to the constraints that $V_1, V_2, \ldots, V_t \geqslant 0$.
              Let $V_1^*, \ldots, V_t^*$ denote the optimal solution to the above
              optimization problem, and let $f^*$ be the corresponding
              minimum value of the objective function $f$.
              If $f^* < f_{opt}$, do:
              { $f_{opt} \leftarrow f^*$.
                    $\vec{V}_{opt} \leftarrow (V_1^*, \ldots, V_t^*)$.
                    For $d = 1, 2, \ldots, 2^t$
                    { For $q = k_{d-1}(j) + 1, k_{d-1}(j) + 2, \ldots, k_d(j)$
                              { $\left(b_{q,1}^{opt}, b_{q,2}^{opt}, \ldots, b_{q,t}^{opt}\right)^T \leftarrow B_d^{\pi_i}$.
} } } } }
Output the solution $\vec{V}_{opt}, b_{1,1}^{opt}, \ldots, b_{n,t}^{opt}$. (The corresponding
minimized value of the cost function $\mathcal{C}_p(\Theta, L)$ is $f_{opt}^{1/p}$.
```

Theorem 6.3.4. *Algorithm* 7.5.1 *outputs an optimal solution to the parallel programming problem.*

Proof. In each iteration of Algorithm 7.5.1, $V_1^*, \ldots, V_t^*$ and the values $b_{1,1}, \ldots, b_{n,t}$ (which are specified by the vectors $B_d^{\pi_i}$ in the iteration) form a feasible solution to the parallel programming problem. On the other hand, an optimal solution to the parallel programming problem that has the properties in Lemma 6.3.1 and Lemma 6.3.2 must be found by Algorithm 7.5.1. ■

It can be seen that when $p = 1$, 2, and ∞ (which are the most commonly used metrics for $\mathcal{C}_p(\Theta, L)$), the time complexity of Algorithm 7.5.1 is polynomial in n. (Note that t is a small constant.) For example, consider $p = 2$. In each iteration of Algorithm 7.5.1, the optimization problem is a quadratic programming problem with $t = O(1)$ non-negativity constraints for variables, which can be solved with time complexity of $O(n)$ [10]. In general, for any p we get a polynomial with a fixed number of variables, t, and finding its minimum can be done with time complexity of $O(n)$. There are $2^t!\binom{n+2^t-1}{2^t-1} = O(n^{2^t-1})$ iterations. So Algorithm 7.5.1 has time complexity $O(n^{2^t})$.

Note that we can slightly optimize the complexity of Algorithm 7.5.1 by using the fact that not all the $2^t!$ permutations π_i need to be checked. Since we require that for each permutation $\mathcal{I}(\vec{V})$,

$$\vec{V} \cdot B_1^{\pi_{\mathcal{I}(\vec{V})}} \leqslant \vec{V} \cdot B_2^{\pi_{\mathcal{I}(\vec{V})}} \leqslant \cdots \leqslant \vec{V} \cdot B_{2^t}^{\pi_{\mathcal{I}(\vec{V})}},$$

if we assume without loss of generality that

$$V_1 < V_2 < \cdots < V_t$$

(the programming order does not matter since there is no feedback), then for any permutation, $\mathcal{I}(\vec{V})$,

$$B_1^{\pi_{\mathcal{I}(\vec{V})}} = (0, \dots, 0), B_2^{\pi_{\mathcal{I}(\vec{V})}} = (1, 0, \dots, 0), B_3^{\pi_{\mathcal{I}(\vec{V})}} = (0, 1, 0, \dots, 0)$$

and

$$B_{2^t}^{\pi_{\mathcal{I}(\vec{V})}} = (1, \dots, 1), B_{2^t-1}^{\pi_{\mathcal{I}(\vec{V})}} = (0, 1, \dots, 1), B_{2^t-2}^{\pi_{\mathcal{I}(\vec{V})}} = (1, 0, 1, \dots, 1).$$

However, this does not improve the complexity order of the algorithm as the complexity significantly results from the second for-loop which we cannot optimize. In the next example, we demonstrate how to apply Algorithm 7.5.1 for $t = 3$.

Example 6.3.1. Let us demonstrate how Algorithm 7.5.1 works for $t = 3$. Again, we assume without loss of generality that $0 < V_1 < V_2 < V_3$. Therefore,

$$0 < V_1 < V_2 < V_3, V_1 + V_2 < V_1 + V_3 < V_2 + V_3 < V_1 + V_2 + V_3$$

and so there are two permutations of the vectors in $\{0, 1\}^3$ that we need to consider. The first loop in Algorithm 7.5.1 goes through two possible permutations and the second loop goes through all possible set of integers $\{k_0, k_1, \dots, k_7, k_8\}$, such that

$$1 = k_0 \leqslant k_1 \leqslant \cdots \leqslant k_7 \leqslant k_8 = n + 1,$$

and there are $\binom{n+8-1}{8-1} = \binom{n+7}{7}$ ways to choose these numbers. For example, assume that $V_3 < V_1 + V_2$, then for each set of such integers $1 = k_0 \leqslant k_1 \leqslant \cdots \leqslant k_7 \leqslant k_8 = n+1$ we find the solution of V_1, V_2, V_3 that minimizes the cost function

$$\sum_{q=k_0}^{k_1-1} \theta_q^2 + \sum_{q=k_1}^{k_2-1} (\theta_q - V_1)^2 + \sum_{q=k_2}^{k_3-1} (\theta_q - V_2)^2 + \sum_{q=k_3}^{k_4-1} (\theta_q - V_3)^2$$
$$+ \sum_{q=k_4}^{k_5-1} (\theta_q - (V_1 + V_2))^2 + \sum_{q=k_5}^{k_6-1} (\theta_q - (V_1 + V_3))^2$$
$$+ \sum_{q=k_6}^{k_7-1} (\theta_q - (V_2 + V_3))^2 + \sum_{q=k_7}^{k_8-1} (\theta_q - (V_1 + V_2 + V_3))^2.$$

Finally, we are only left to output the solution with the minimum cost function.

We conclude this section by studying a simplified yet useful case of parallel programming. In this case, all n cells have the same target cell level, that is, $\theta_1 = \cdots = \theta_n = \theta$. (For multi-level cells (MLC), it is a natural heuristic solution to program cells of the same target level together, which corresponds to this case.) We consider programming noise ϵ; that is, if the program voltage V is applied to a cell c_i, its charge level will increase by $\alpha_i V + \epsilon_i(V)$, where the programming noise $\epsilon_i(V)$ is a random number related to V. The values of $\alpha_1, \ldots, \alpha_n$ are unknown, but we see them as i.i.d. random variables with known expectation $\mu_1(\alpha)$ and second moment $\mu_2(\alpha)$. (That is, if we use $E(x)$ to denote the expectation of a random variable x, then for $i \in [n]$, $E(\alpha_i) = \mu_1(\alpha)$ and $E(\alpha_i^2) = \mu_2(\alpha)$. Note that the statistics $\mu_1(\alpha)$ and $\mu_2(\alpha)$ are easy to obtain by the memory by testing many cells with the values of the injected and trapped charges in the cells.) We also assume that the programming noise $\epsilon_1(V), \ldots, \epsilon_n(V)$ are i.i.d. random variables; and for $i \in [n]$, we assume $E(\epsilon_i(V)) = 0$ (namely, its expectation is zero) and $E\left((\epsilon_i(V))^2\right) = \sigma V^2$ (namely, its standard deviation is proportional to the program voltage V). We can use t rounds of programming. We consider the ℓ_2 metric for the cost function and define it as

$$\mathcal{C} \triangleq \sum_{i=1}^{n} (\theta - \ell_{i,t})^2 .$$

And our objective is to minimize $E(\mathcal{C})$, that is, the expected cost. We call this case *uniform programming*.

Clearly, the optimal solution is to apply the same set of program voltages $V_1, \ldots, V_t$ to all the n cells. The following theorem presents the optimal solution and the optimal cost.

Theorem 6.3.5. *For the uniform programming problem, the optimal solution is to set*

$$V_1 = \cdots = V_t = \frac{\theta \mu_1(\alpha)}{t \mu_2(\alpha) + \sigma}$$

and $b_{i,j} = 1$ *for* $i \in [n]$ *and* $j \in [t]$. *The corresponding minimized value of the expected cost* $E(\mathcal{C})$ *is*

$$n\left(1 - \frac{\mu_1(\alpha)^2}{\mu_2(\alpha) + \frac{\sigma}{t}}\right)\theta^2.$$

Proof. Since there is no interference between the cells $\ell_{1,t}, \ell_{2,t}, \ldots, \ell_{n,t}$ are i.i.d. random variables. So $E(\mathcal{C}) = nE\left((\theta - \ell_{1,t})^2\right)$. For $i \in [t]$, let $\epsilon_{1,i}$ denote the programming noise for cell c_1 in the i-th round of programming. Then,

$$\begin{aligned}E(\mathcal{C})/n &= E\left(\left(\theta - \textstyle\sum_{i=1}^t (\alpha_1 V_i + \epsilon_{1,i})\right)^2\right)\\ &= \theta^2 - 2\theta E\left(\textstyle\sum_{i=1}^t (\alpha_1 V_i + \epsilon_{1,i})\right) + E\left(\left(\textstyle\sum_{i=1}^t (\alpha_1 V_i + \epsilon_{1,i})\right)^2\right)\\ &= \theta^2 - 2\theta\mu_1(\alpha)\textstyle\sum_{i=1}^t V_i + \mu_2(\alpha)\left(\textstyle\sum_{i=1}^t V_i\right)^2 + \sigma\textstyle\sum_{i=1}^t V_i^2.\end{aligned}$$

Note that $\sum_{i=1}^t V_i^2 \geqslant \frac{1}{t}\left(\sum_{i=1}^t V_i\right)^2$. So we get

$$E(\mathcal{C})/n \geqslant \left(\mu_2(\alpha) + \frac{\sigma}{t}\right)\left(\sum_{i=1}^t V_i\right)^2 - 2\theta\mu_1(\alpha)\sum_{i=1}^t V_i + \theta^2 \geqslant \left(1 - \frac{\mu_1(\alpha)^2}{\mu_2(\alpha) + \frac{\sigma}{t}}\right)\theta^2.$$

The above inequalities will become equalities if and only if

$$\sum_{i=1}^t V_i = \frac{\theta\mu_1(\alpha)}{\mu_2(\alpha) + \frac{\sigma}{t}}$$

and $V_1 = V_2 = \cdots = V_t$. ■

6.4 Programming with Inter-Cell Interference

In order to improve the capacity of flash memories, their cell size is getting smaller and smaller. One of the main side effects of scaling down the flash memory technology is an increase in the inter-cell interference, also known as cell-to-cell coupling [1, 9]. While programming a flash memory cell, its neighbors are also affected by the charge that is injected into the cell, which causes inter-cell interference [5, 14]. There are several ways to overcome inter-cell interference, such as powerful error-correcting codes or modulation codes. In this section, we study how to overcome this problem by adjusting the cell programming algorithms so the inter-cell interference is taken into account and thus is reduced. We show how to modify Algorithm 7.5.1 in Section 6.3 in case there is inter-cell interference while the cells are being programmed. For simplicity of presentation, we will only consider the ℓ_2 metric, but modifications for other metrics are straightforward. Also, we consider inter-cell interference only in one dimension.

The problem formulation is similar to the description in Section 6.2. Let $V_1, V_2, \ldots, V_t$ and $b_{i,j}$ for $i \in [n], j \in [t]$ be as defined previously. For $i \in [n], j \in [t]$, let $y_{i,j}$ be the amount of charge that is injected to the i-th cell after the j-th programming round, that is,

$$y_{i,j} = (V_1, V_2, \ldots, V_j) \cdot (b_{i,1}, b_{i,2}, \ldots, b_{i,j})^T.$$

Then, $\ell_{i,j}$, the level of c_i after the j-th programming round is given by

$$\ell_{i,j} = \alpha_i y_{i,j} + \alpha_i \beta (y_{i-1,j} + y_{i+1,j}),$$

where β is a small constant indicating the amount of inter-cell interference. By convention, we define for all $j \in [t]$, $y_{0,j} = y_{n+1,j} = 0$.

Assume that the target levels are $\Theta = (\theta_1, \ldots, \theta_n)$. First, we define the modified target levels $\Theta' = (\theta'_1, \ldots, \theta'_n)$ which satisfy

$$\theta'_i = \theta_i - \alpha_i \beta \left(\frac{\theta'_{i-1}}{\alpha_{i-1}} + \frac{\theta'_{i+1}}{\alpha_{i+1}} \right) \tag{6.1}$$

for $i \in [n]$ and $\theta'_0 = \theta'_{n+1} = 0$, by convention. We assume that β is small enough such that $\theta'_i \geqslant 0$, for $i \in [n]$. Let the matrix A be

$$A_n = \begin{pmatrix} 1 & \frac{\alpha_1 \beta}{\alpha_2} & 0 & 0 & \cdots & 0 \\ \frac{\alpha_2 \beta}{\alpha_1} & 1 & \frac{\alpha_2 \beta}{\alpha_3} & 0 & \cdots & 0 \\ 0 & \frac{\alpha_3 \beta}{\alpha_2} & 1 & \frac{\alpha_3 \beta}{\alpha_4} & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 & \frac{\alpha_n \beta}{\alpha_{n-1}} & 1 \end{pmatrix},$$

then equation (6.1) can be expressed also in the following way

$$A_n \cdot (\theta'_1, \ldots, \theta'_n)^T = (\theta_1, \ldots, \theta_n)^T.$$

Lemma 6.4.1. *If $\beta < \sqrt{\frac{1}{n-1}}$, then there is a unique solution to the values of θ'_i for $i \in [n]$.*

Proof. We need to show that the matrix A_n is invertible. We show by induction that $1 - (n-1)\beta^2 \leqslant |A_n| \leqslant 1$. The base of the induction is the matrix

$$A_2 = \begin{pmatrix} 1 & \frac{\alpha_1 \beta}{\alpha_2} \\ \frac{\alpha_2 \beta}{\alpha_1} & 1 \end{pmatrix},$$

and its determinant satisfies

$$|A_2| = 1 - \frac{\alpha_1 \beta}{\alpha_2} \cdot \frac{\alpha_2 \beta}{\alpha_1} = 1 - \beta^2,$$

and $|A_2| \leqslant 1$. Next assume that for $i \leqslant n-1$,

$$1-(i-1)\beta^2 \leqslant |A_i| \leqslant 1.$$

Note that

$$|A_n| = |A_{n-1}| - \frac{\alpha_n \beta}{\alpha_{n-1}} \cdot \frac{\alpha_{n-1}\beta}{\alpha_n} \cdot |A_{n-2}| \leqslant |A_{n-1}| \leqslant 1$$

and

$$|A_n| = |A_{n-1}| - \beta^2 \cdot |A_{n-2}| \geqslant 1-(n-2)\beta^2 - \beta_2 = 1-(n-1)\beta^2.$$

Therefore, $|A_n| \geqslant 1-(n-1)\beta^2$ and since $\beta < \sqrt{\frac{1}{n-1}}$, we get that $|A_n| > 0$, that is, the matrix A_n is invertible and there is a unique solution to the values of θ'_i for $i \in [n]$. ■

In the rest of this section we will assume that $\beta < \sqrt{\frac{1}{n-1}}$, so there is a unique solution to the values of θ'_i for $i \in [n]$ and they are all positive.

Next, we seek to apply Algorithm 7.5.1 with the values of θ'_i, $i \in [n]$. However, we need to apply the following modifications. First, we sort, as before, the cells and let $\{p_1, \ldots, p_n\}$ be a permutation of $\{1, \ldots, n\}$ such that

$$\frac{\theta'_{p_1}}{\alpha_{p_1}} < \frac{\theta'_{p_2}}{\alpha_{p_2}} < \cdots < \frac{\theta'_{p_n}}{\alpha_{p_n}}.$$

Note that now the order of the cells does matter because of the inter-cell interference. We assume here that all these ratios are different since if there are two identical values then we can treat them as the same cell. And let

$$\epsilon = \min_{i=1,\ldots,n-1} \left\{ \frac{\theta'_{p_{i+1}}}{\alpha_{p_{i+1}}} - \frac{\theta'_{p_i}}{\alpha_{p_i}} \right\}.$$

Also, the function that we minimize in Algorithm 7.5.1 is

$$f = \sum_{d=1}^{2^t} \sum_{q=k_{d-1}(p_j)+1, k_{d-1}(p_j)+2, \ldots, k_d(p_j)} \left| \theta_q - \alpha_q y_q - \alpha_q \beta (y_{q-1} + y_{q+1}) \right|^p,$$

where for all $1 \leqslant i \leqslant n$, y_i is the amount of charge that is injected to the i-th cell, i.e., in Algorithm 7.5.1

$$y_i = (V_1, \ldots, V_t) \cdot B_d^{\pi_i}.$$

The rest of the Algorithm remains the same and we denote the new algorithm by Algorithm 7.5.1A. The new function f which we need to minimize is again a polynomial in t variables which can be solved in time complexity $O(n)$. Therefore, the complexity of Algorithm 7.5.1A does not change.

Let us prove the conditions where Algorithm 7.5.1A results with the optimal solution. The main idea is to prove the equivalent of Lemma 6.3.1. Assume that an optimal solution is given by the choice of V_i for $i \in [t]$ and $b_{i,j}$ for $i \in [t], j \in [n]$. The cost of this solution is

$$\delta = \sum_{i=1}^{n} \left(\theta_i - \alpha_i(y_i + \beta(y_{i-1} + y_{i+1}))\right)^2,$$

where $y_i = (V_1, \ldots, V_t) \cdot (b_{i,1}, \ldots, b_{i,t})$.

Lemma 6.4.2. *If the cost of an optimal solution is δ then*

$$\sum_{i=1}^{n} (\theta'_i - \alpha_i y_i)^2 \leqslant \frac{\delta}{1-(n-1)\beta^2}.$$

Proof. Note the following relation:

$$\begin{aligned}\theta_i - \alpha_i(y_i + \beta(y_{i-1} + y_{i+1})) &= \theta_i - \alpha_i\beta\left(\frac{\theta'_{i-1}}{\alpha_{i-1}} + \frac{\theta'_{i+1}}{\alpha_{i+1}}\right)\\ &- \alpha_i y_i + \alpha_i\beta\left(\frac{\theta'_{i-1}}{\alpha_{i-1}} + \frac{\theta'_{i+1}}{\alpha_{i+1}}\right) - \alpha_i\beta(y_{i-1} + y_{i+1})\\ &= \theta'_i - \alpha_i y_i + \alpha_i\beta\left(\frac{\theta'_{i-1} - \alpha_{i-1}y_{i-1}}{\alpha_{i-1}} + \frac{\theta'_{i+1} - \alpha_{i+1}y_{i+1}}{\alpha_{i+1}}\right).\end{aligned}$$

If we define for $i \in [n]$, $a_i = \theta_i - \alpha_i(y_i + \beta(y_{i-1} + y_{i+1}))$ and $b_i = \theta'_i - \alpha_i y_i$, then we get the following equations:

$$a_i = b_i + \alpha_i\beta\left(\frac{b_{i-1}}{\alpha_{i-1}} + \frac{b_{i+1}}{\alpha_{i+1}}\right),$$

which can be written in the form

$$A_n \cdot (b_1, \ldots, b_n)^T = (a_1, \ldots, a_n)^T.$$

Hence, since the matrix A_n is invertible $(b_1, \ldots, b_n)^T = A_n^{-1} \cdot (a_1, \ldots, a_n)^T$, and so

$$||(b_1, \ldots, b_n)||_2 = |A_n^{-1}| \cdot ||(a_1, \ldots, a_n)||_2 \leqslant \frac{\delta}{1-(n-1)\beta^2}.$$

■

In particular, we get from Lemma 6.4.2 that for all $i \in [n]$,

$$|\theta'_i - \alpha_i y_i| \leqslant \sqrt{\frac{\delta}{1-(n-1)\beta^2}}.$$

Lemma 6.4.3. *If the cost of an optimal solution is δ and $\sqrt{\frac{\delta}{1-(n-1)\beta^2}} < \frac{\epsilon}{4\beta}$, then for $1 \leqslant i < j \leqslant n$, $y_{p_i} \leqslant y_{p_j}$.*

Proof. Let us denote $\sqrt{\frac{\delta}{1-(n-1)\beta^2}}$ by δ. For $1 \leqslant i \leqslant n$, let $\ell_i = \alpha_i(y_i + \beta(y_{i-1} + y_{i+1}))$ and note that the cost function is monotonically increasing in $|\theta_i - \ell_i|$. Assume to the contrary that there exist $1 \leqslant i < j \leqslant n$ such that $y_{p_i} > y_{p_j}$. Since $|\theta_{p_i} - \ell_{p_i}|$ is minimized we get that

$$
\begin{aligned}
&|\theta_{p_i} - \alpha_{p_i}(y_{p_i} + \beta(y_{p_i-1} + y_{p_i+1}))| \leqslant |\theta_{p_i} - \alpha_{p_i}(y_{p_j} + \beta(y_{p_i-1} + y_{p_i+1}))|,\\
&\alpha_{p_i}^2(y_{p_i}^2 - y_{p_j}^2) \leqslant 2\alpha_{p_i}(y_{p_i} - y_{p_j})(\theta_{p_i} - \beta(y_{p_i-1} + y_{p_i+1})),\\
&\alpha_{p_i}(y_{p_i} + y_{p_j}) \leqslant 2(\theta_{p_i} - \beta(y_{p_i-1} + y_{p_i+1})).
\end{aligned}
$$

and similarly we get

$$\alpha_{p_j}(y_{p_i} + y_{p_j}) \geqslant 2(\theta_{p_j} - \beta(y_{p_j-1} + y_{p_j+1})).$$

Thus,

$$\frac{\theta_{p_i} - \beta(y_{p_i-1} + y_{p_i+1})}{\alpha_{p_i}} \geqslant \frac{\theta_{p_j} - \beta(y_{p_j-1} + y_{p_j+1})}{\alpha_{p_j}}.$$

Since $\frac{\theta'_{p_i}}{\alpha_{p_i}} < \frac{\theta'_{p_j}}{\alpha_{p_j}}$ we get that

$$\frac{\theta_{p_i} - \alpha_{p_i}\beta(\theta'_{p_i-1} + \theta'_{p_i+1})}{\alpha_{p_i}} < \frac{\theta_{p_j} - \alpha_{p_j}\beta(\theta'_{p_j-1} + \theta'_{p_j+1})}{\alpha_{p_j}}.$$

Together these imply

$$
\begin{aligned}
&\frac{\theta_{p_j} - \alpha_{p_j}\beta(y_{p_j-1} + y_{p_j+1})}{\alpha_{p_j}}\\
&\geqslant^{(1)} \frac{\theta_{p_j} - \alpha_{p_j}\beta(\theta'_{p_j-1}/\alpha_{p_j-1} + \theta'_{p_j+1}/\alpha_{p_j+1} + 2\delta')}{\alpha_{p_j}}\\
&= \frac{\theta'_{p_j}}{\alpha_{p_j}} - 2\delta'\beta >^{(2)} \frac{\theta'_{p_i}}{\alpha_{p_i}} - 2\delta'\beta + \epsilon\\
&= \frac{\theta_{p_i} - \alpha_{p_i}\beta(\theta'_{p_i-1}/\alpha_{p_i-1} + \theta'_{p_i+1}/\alpha_{p_i+1} + 2\delta' - \epsilon/\beta)}{\alpha_{p_i}}\\
&\geqslant^{(3)} \frac{\theta_{p_i} - \alpha_{p_i}\beta(\theta'_{p_i-1}/\alpha_{p_i-1} + \theta'_{p_i+1}/\alpha_{p_i+1} - 2\delta')}{\alpha_{p_i}}\\
&\geqslant^{(4)} \frac{\theta_{p_i} - \alpha_{p_i}\beta(y_{p_i-1} + y_{p_i+1})}{\alpha_{p_i}},
\end{aligned}
$$

where (1) and (4) result from the conclusion that for all $i \in [n]$, $|\theta'_i - \alpha_i y_i| \leqslant \delta'$. Inequality (2) results from the definition of ϵ and inequality (3) results from the condition in the lemma that $\delta' < \frac{\epsilon}{4\beta}$. Hence, we get a contradiction. ■

To conclude, we prove the following Theorem.

Theorem 6.4.4. *If* $\sqrt{\frac{\delta}{1-(n-1)\beta^2}} \leqslant \frac{\epsilon}{4\beta}$, *then Algorithm* 7.5.1*A outputs an optimal solution.*

Proof. According to Lemma 6.4.3, for any optimal solution the amount of charge that is injected to the cells is non-decreasing according to the permutation of the cell $p_1, p_2, \ldots, p_n$. Therefore, this order of injecting the cells is a feasible solution of Algorithm 7.5.1A and hence can be found by the algorithm. ■

6.5 Feedback Information on Cell Levels

In this section, we extend the study to parallel programming with feedback information on cell levels. That is, the memory can measure the cell levels after every round of charge injection. Note that in practice, it is very time consuming to measure the exact level of every cell. It is much faster to compare the cell levels (in parallel using comparators) with one or more preset threshold levels, to see if the cell levels are above or below the threshold level [3, 13, 18]. This is the scheme we consider here. By obtaining this feedback information on cell levels, the memory can learn more about the cells' hardness for charge injection, and adaptively choose the subsequent program voltages for optimal performance.

Let $\alpha_{\min}$ and $\alpha_{\max}$ be two known parameters, where $0 < \alpha_{\min} < \alpha_{\max}$. We assume that initially (i.e., before programming starts), the only knowledge on cells' hardness for charge injection is that for $i \in [n]$, $\alpha_i \in [\alpha_{\min}, \alpha_{\max}]$. After every round of programming, based on the feedback information on cell levels, the memory can estimate the values of the α_i's better, and adaptively optimize the following program voltages. Therefore, the programming algorithm is a combination of two iterative processes: *iteratively obtaining information on the cells' hardness for programming (i.e., the values of α_i's)*, and *adaptively optimizing the program voltages (i.e., $V_1, \ldots, V_t$) and the on/off of cells (i.e., $b_{1,1}, \ldots, b_{n,t}$).*

In this section, we focus on the first iterative process, and study this related problem: How much information can be learned about a cell's hardness for charge injection, α, through t rounds of programming? Specifically, let $[\alpha'_{\min}, \alpha'_{\max}] \subseteq [\alpha_{\min}, \alpha_{\max}]$ denote the range we can narrow down to such that after t rounds of programming, we can learn for sure that $\alpha \in [\alpha'_{\min}, \alpha'_{\max}]$. The smaller $\alpha'_{\max} - \alpha'_{\min}$ is, the better.

We assume that after every round of programming, we can compare a cell with one preset threshold level. We formally formulate the problem as follows. Let c be a cell whose initial level is 0. Let α denote the cell's hardness for charge injection, such that when a program voltage

V is applied, the cell's level will increase by αV. Initially, the only knowledge about α is that $\alpha \in [\alpha_{\min}, \alpha_{\max}]$ for some known parameters $\alpha_{\min}, \alpha_{\max}$. We can use up to t rounds of programming (whose program voltages are denoted by $V_1, \ldots, V_t$), and choose in advance r threshold levels $\tau_1, \tau_2, \ldots, \tau_r$. For $i \in [t]$, let ℓ_i denote the cell's level after the i-th round of programming, and let $[\alpha_i^{\min}, \alpha_i^{\max}]$ denote the range such that after the i-th round of programming, we know for sure that $\alpha \in [\alpha_i^{\min}, \alpha_i^{\max}]$. (By convention, let $\ell_0 = 0$ and $[\alpha_0^{\min}, \alpha_0^{\max}] = [\alpha_{\min}, \alpha_{\max}]$. Clearly, when $0 \leqslant i < j \leqslant t$, $[\alpha_i^{\min}, \alpha_i^{\max}] \supseteq [\alpha_j^{\min}, \alpha_j^{\max}]$.) For $i \in [t]$, after the i-th round of programming, we can compare ℓ_i with one threshold level – say τ_j – to see if $\ell_i < \tau_j$ or $\ell_i \geqslant \tau_j$, compute $\alpha_i^{\min}$ and $\alpha_i^{\max}$, and adaptively choose the next program voltage V_{i+1}. Our objective is to choose $\tau_1, \ldots, \tau_r$ in advance, choose $V_1, \ldots, V_t$ online, such that $\alpha_t^{\max} - \alpha_t^{\min}$ is minimized (in the worst case).

Let $\Delta(t, r, \alpha_{\min}, \alpha_{\max})$ denote the smallest achievable value of $\alpha_t^{\max} - \alpha_t^{\min}$ over all possible solutions. Intuitively, $\Delta(t, r, \alpha_{\min}, \alpha_{\max}) \geqslant (\alpha_{\max} - \alpha_{\min})/2^t$. (Every comparison with a threshold level can reduce the interval size by at most half.) We now present a better bound. Given $i \geqslant j$, define

$$\left(\binom{i}{j}\right) \triangleq \sum_{k=0}^{j} \binom{i}{k}$$

and for $i < j$, define $\left(\binom{i}{j}\right) \triangleq 2^i$. Note that

$$\left(\binom{i}{j}\right) = \left(\binom{i-1}{j}\right) + \left(\binom{i-1}{j-1}\right).$$

Theorem 6.5.1. *For all $r, t \geqslant 0$,*

$$\Delta(t, r, \alpha_{\min}, \alpha_{\max}) \geqslant \frac{(\alpha_{\max} - \alpha_{\min})}{\left(\binom{t}{r}\right)}.$$

Proof. We prove by induction on t that for all $r \geqslant 0$,

$$\Delta(t, r, \alpha_{\min}, \alpha_{\max}) \geqslant (\alpha_{\max} - \alpha_{\min}) \Big/ \left(\binom{t}{r}\right).$$

For the base case where $t = 0$, for all $r \geqslant 0$,

$$\Delta(0, r, \alpha_{\min}, \alpha_{\max}) = \alpha_{\max} - \alpha_{\min}.$$

Let $t \geqslant 1$ and assume that the claim is true for $t-1$. On the first round of programming, we compare the cell level ℓ with some threshold level τ; then for some $\alpha' \in [\alpha_{\min}, \alpha_{\max}]$ we can determine if $\alpha \geqslant \alpha'$ (because $\ell \geqslant \tau$) or $\alpha < \alpha'$ (because $\ell < \tau$). There are $t-1$ more rounds; and if the cell level ℓ has exceeded τ, there are at most $r-1$ threshold levels left to compare with. (Note that the cell level can only increase.) So by induction, we get

$$\begin{aligned}
&\Delta(t, r, \alpha_{\min}, \alpha_{\max}) \\
&= \min_{\alpha'} \max \left\{ \Delta(t-1, r-1, \alpha', \alpha_{\max}), \Delta(t-1, r, \alpha_{\min}, \alpha') \right\} \\
&\geqslant \min_{\alpha'} \max \left\{ \frac{\alpha_{\max} - \alpha'}{\left(\binom{t-1}{r-1}\right)}, \frac{\alpha' - \alpha_{\min}}{\left(\binom{t-1}{r}\right)} \right\}.
\end{aligned}$$

If $\alpha' \geqslant \frac{\left(\binom{t-1}{r}\right)\alpha_{\max} + \left(\binom{t-1}{r-1}\right)\alpha_{\min}}{\left(\binom{t}{r}\right)}$, then

$$\begin{aligned}
\frac{\alpha' - \alpha_{\min}}{\left(\binom{t-1}{r}\right)} &\geqslant \frac{\frac{\left(\binom{t-1}{r}\right)\alpha_{\max} + \left(\binom{t-1}{r-1}\right)\alpha_{\min}}{\left(\binom{t}{r}\right)} - \alpha_{\min}}{\left(\binom{t-1}{r}\right)} \\
&= \frac{\left(\binom{t-1}{r}\right)\alpha_{\max} - \left(\binom{t-1}{r}\right)\alpha_{\min}}{\left(\binom{t}{r}\right) \cdot \left(\binom{t-1}{r}\right)} = \frac{\alpha_{\max} - \alpha_{\min}}{\left(\binom{t}{r}\right)}.
\end{aligned}$$

Similarly, if $\alpha' \leqslant \frac{\left(\binom{t-1}{r}\right)\alpha_{\max} + \left(\binom{t-1}{r-1}\right)\alpha_{\min}}{\left(\binom{t}{r}\right)}$, then

$$\begin{aligned}
\frac{\alpha_{\max} - \alpha'}{\left(\binom{t-1}{r-1}\right)} &\geqslant \frac{\alpha_{\max} - \frac{\left(\binom{t-1}{r}\right)\alpha_{\max} + \left(\binom{t-1}{r-1}\right)\alpha_{\min}}{\left(\binom{t}{r}\right)}}{\left(\binom{t-1}{r-1}\right)} \\
&= \frac{\left(\binom{t-1}{r-1}\right)\alpha_{\max} - \left(\binom{t-1}{r-1}\right)\alpha_{\min}}{\left(\binom{t}{r}\right) \cdot \left(\binom{t-1}{r-1}\right)} = \frac{\alpha_{\max} - \alpha_{\min}}{\left(\binom{t}{r}\right)}.
\end{aligned}$$

Therefore, we get $\Delta(t, r, \alpha_{\min}, \alpha_{\max}) \geqslant \frac{\alpha_{\max} - \alpha_{\min}}{\left(\binom{t}{r}\right)}$. ■

We now present an optimal algorithm whose performance matches the above bound. It has $r \leqslant t$. (Having $r > t$ does not help since we compare with only one threshold level in every round of programming.) In the algorithm, for $i = 1, \ldots, r$, let $\tau_i = \tau_1 \left(\frac{\alpha_{\max}}{\alpha_{\min}}\right)^{i-1}$. (The value of $\tau_1 > 0$ can be arbitrary. By convention, let $\tau_0 \triangleq 0$.) Let $z_1, \ldots, z_{t+1}$ and *flag* be integer parameters, and set $z_1 = r$ and *flag* $= 1$. Then, for $i = 1, 2, \ldots, t$, the i-th programming round (and its following computation) is performed as follows:

1. If $z_i = 0$, then $\alpha_i^{\min} \leftarrow \alpha_{i-1}^{\min}$, $\alpha_i^{\max} \leftarrow \alpha_{i-1}^{\max}$, $z_{i+1} \leftarrow 0$ and skip the next three steps. (In this case, we see V_i as $V_i = 0$; namely, the cell is not really programmed.)

2. $\alpha_i' \leftarrow \frac{\alpha_{i-1}^{\min}\left(\binom{t-i}{z_i-1}\right)+\alpha_{i-1}^{\max}\left(\binom{t-i}{z_i}\right)}{\left(\binom{t-i}{z_i-1}\right)+\left(\binom{t-i}{z_i}\right)}$.

3. If *flag* $= 1$, then $V_i \leftarrow \frac{\tau_{r-z_i+1}}{\alpha_i'} - \frac{\tau_{r-z_i}}{\alpha_{i-1}^{\min}}$;

   If *flag* $= 0$, then $V_i \leftarrow \frac{\tau_{r-z_i+1}}{\alpha_i'} - \frac{\tau_{r-z_i+1}}{\alpha_{i-1}^{\max}}$.

   Program the cell with program voltage V_i.

4. Compare cell level ℓ_i with threshold level τ_{r-z_i+1}.

   If $\ell_i < \tau_{r-z_i+1}$, then $\alpha_i^{\min} \leftarrow \alpha_{i-1}^{\min}$, $\alpha_i^{\max} \leftarrow \alpha_i'$, $flag \leftarrow 0$, $z_{i+1} \leftarrow z_i$.

   If $\ell_i \geqslant \tau_{r-z_i+1}$, then $\alpha_i^{\min} \leftarrow \alpha_i'$, $\alpha_i^{\max} \leftarrow \alpha_{i-1}^{\max}$, $z_{i+1} \leftarrow z_i - 1$.

The next lemma shows that the program voltages are all non-negative, which means the algorithm can be successfully implemented.

Lemma 6.5.2. *In the algorithm, for* $i = 1, \ldots, t$, $V_i \geqslant 0$.

Proof. For all $1 \leqslant i \leqslant t$, if $z_i = 0$ then $V_i = 0$ and the assertion is correct. For $i = 1$,

$$V_1 = \frac{\tau_1}{\alpha_1'} - \frac{\tau_0}{\alpha_{i-1}^{\min}} = \frac{\tau_1}{\alpha_1'} \geqslant 0.$$

Now consider $i \geqslant 2$ and $z_i > 0$. If $flag = 1$, then

$$V_i = \frac{\tau_{r-z_i+1}}{\alpha_i'} - \frac{\tau_{r-z_i}}{\alpha_{i-1}^{\min}} = \frac{\alpha_{\max}}{\alpha_{\min}} \cdot \frac{\tau_{r-z_i}}{\alpha_i'} - \frac{\tau_{r-z_i}}{\alpha_{i-1}^{\min}} = \tau_{r-z_i}\left(\frac{\alpha_{\max}}{\alpha_{\min}} \cdot \frac{1}{\alpha_i'} - \frac{1}{\alpha_{i-1}^{\min}}\right).$$

Since $\alpha_{\min} \leqslant \alpha_{i-1}^{\min}$ and $\alpha_i' \leqslant \alpha_{\max}$, we get $V_i \geqslant 0$. Similarly, if $flag = 0$, then

$$V_i = \frac{\tau_{r-z_i+1}}{\alpha_i'} - \frac{\tau_{r-z_i+1}}{\alpha_{i-1}^{\max}} = \tau_{r-z_i+1} \cdot \left(\frac{1}{\alpha_i'} - \frac{1}{\alpha_{i-1}^{\max}}\right) \geqslant 0.$$

■

The following lemma proves a property of the total programmed voltage after each iteration.

Lemma 6.5.3. *In the algorithm, for* $i = 1, \ldots, t$, *we have* $\sum_{j=1}^{i} V_j = (\tau_{r-z_i+1})/\alpha_i'$ *when* $z_i > 0$. *(If* $z_i = 0$, $V_i = 0$.*)*

Proof. We prove the lemma by induction on i. On the first programming round $z_1 = r$ and the program voltage,

$$V_1 = \frac{\tau_1}{\alpha'_1} - \frac{\tau_0}{\alpha_0^{\min}} = \frac{\tau_1}{\alpha'_1},$$

is also the total programmed voltage. Assume that the assertion is correct for $i-1 < t$, so the total programmed voltage is $\frac{\tau_{r-z_{i-1}+1}}{\alpha'_{i-1}}$. We will prove its validity for the i-th programming round. Let us consider the following two cases:

1. If on the $(i-1)$-st round $\ell_{i-1} < \tau_{r-z_{i-1}+1}$ then on the i-th round $\alpha_{i-1}^{\max} = \alpha'_{i-1}, z_i = z_{i-1}$ so the total programmed voltage on the $(i-1)$-st round is

$$\frac{\tau_{r-z_{i-1}+1}}{\alpha'_{i-1}} = \frac{\tau_{r-z_i+1}}{\alpha_{i-1}^{\max}}$$

and the value of $flag$ is zero. Therefore, the program voltage on the i-th round is

$$V_i = \frac{\tau_{r-z_i+1}}{\alpha'_i} - \frac{\tau_{r-z_i+1}}{\alpha_{i-1}^{\max}}$$

and the total programmed voltage is $\frac{\tau_{r-z_i+1}}{\alpha'_i}$.

2. If on the $(i-1)$-st round $\ell_{i-1} \geqslant \tau_{r-z_{i-1}+1}$ then on the i-th round $\alpha_{i-1}^{\min} = \alpha'_{i-1}, z_i = z_{i-1} - 1$ so the total programmed voltage on the $(i-1)$-st round is

$$\frac{\tau_{r-z_{i-1}+1}}{\alpha'_{i-1}} = \frac{\tau_{r-z_i}}{\alpha_{i-1}^{\min}}$$

and the value of $flag$ is one. Therefore, the program voltage on the i-th round is

$$V_i = \frac{\tau_{r-z_i+1}}{\alpha'_i} - \frac{\tau_{r-z_i}}{\alpha_{i-1}^{\min}}$$

so the total programmed voltage is again $\frac{\tau_{r-z_i+1}}{\alpha'_i}$.

■

The next theorem shows the performance of the algorithm.

Theorem 6.5.4. *The algorithm outputs $\alpha_t^{\min}, \alpha_t^{\max}$ such that*

$$\alpha_t^{\max} - \alpha_t^{\min} = \frac{\alpha_{\max} - \alpha_{\min}}{\left(\binom{t}{r}\right)}.$$

Proof. Let k be the smallest integer in $\{1, 2, \ldots, t-1\}$ such that $z_{k+1} = 0$ in the algorithm; if $z_i > 0$ for $i \in [t-1]$, then let $k = t$. The k-th round is the final round when the cell is really programmed, and we have $\alpha_t^{\min} = \alpha_k^{\min}$, $\alpha_t^{\max} = \alpha_k^{\max}$. In the following, we consider only the case $k = t$. (The case $k < t$ is a simple extension.) Note that $z_{t+1} = 0$. We prove for $i \in \{0, 1, \ldots, t\}$,

$$\alpha_t^{\max} - \alpha_t^{\min} = \left(\alpha_i^{\max} - \alpha_i^{\min}\right) / \left(\left(\binom{t-i}{z_{i+1}}\right)\right).$$

This clearly holds for $i = t$. We now prove it using induction on i, for $i = t-1, t-2, \ldots, 0$. (The case $i = 0$ leads to this theorem.)

Consider the $(i+1)$-st round of programming. The parameter α'_{i+1} is chosen such that

$$\alpha_i^{\max} - \alpha'_{i+1} = \frac{\left(\binom{t-i-1}{z_{i+1}-1}\right)\left(\alpha_i^{\max} - \alpha_i^{\min}\right)}{\left(\binom{t-i-1}{z_{i+1}-1}\right) + \left(\binom{t-i-1}{z_{i+1}}\right)}$$

and

$$\alpha'_{i+1} - \alpha_i^{\min} = \frac{\left(\binom{t-i-1}{z_{i+1}}\right)\left(\alpha_i^{\max} - \alpha_i^{\min}\right)}{\left(\binom{t-i-1}{z_{i+1}-1}\right) + \left(\binom{t-i-1}{z_{i+1}}\right)}.$$

This means

$$\frac{\alpha_i^{\max} - \alpha'_{i+1}}{\left(\binom{t-i-1}{z_{i+1}-1}\right)} = \frac{\alpha'_{i+1} - \alpha_i^{\min}}{\left(\binom{t-i-1}{z_{i+1}}\right)}.$$

Since either "$\alpha_{i+1}^{\max} = \alpha_i^{\max}$, $\alpha_{i+1}^{\min} = \alpha'_{i+1}$ and $z_{i+2} = z_{i+1} - 1$" or "$\alpha_{i+1}^{\max} = \alpha'_{i+1}$, $\alpha_{i+1}^{\min} = \alpha_i^{\min}$ and $z_{i+2} = z_{i+1}$", using induction we get

$$\begin{aligned}\alpha_t^{\max} - \alpha_t^{\min} &= \frac{\alpha_{i+1}^{\max} - \alpha_{i+1}^{\min}}{\left(\binom{t-i-1}{z_{i+2}}\right)} = \frac{\alpha_i^{\max} - \alpha'_{i+1}}{\left(\binom{t-i-1}{z_{i+1}-1}\right)} = \frac{\alpha'_{i+1} - \alpha_i^{\min}}{\left(\binom{t-i-1}{z_{i+1}}\right)} \\ &= \frac{\left(\alpha_i^{\max} - \alpha'_{i+1}\right) + \left(\alpha'_{i+1} - \alpha_i^{\min}\right)}{\left(\binom{t-i-1}{z_{i+1}-1}\right) + \left(\binom{t-i-1}{z_{i+1}}\right)} = \frac{\alpha_i^{\max} - \alpha_i^{\min}}{\left(\binom{t-i}{z_{i+1}}\right)}\end{aligned}$$

■

From Theorem 6.5.1 and Theorem 6.5.4, we see that the bound in Theorem 6.5.1 is actually exact. We get the following result.

Corollary 6.5.5. *The above algorithm is optimal. And*

$$\Delta\left(t, r, \alpha_{\min}, \alpha_{\max}\right) = \frac{\alpha_{\max} - \alpha_{\min}}{\left(\binom{t}{r}\right)}.$$

Remark 6.5.1. We note that the solution of this problem is very similar to the egg-drop numbers [2]. The egg-drop number $D_{t,\ell}$ is the the number of floors that one can reach with ℓ eggs and t drops. The egg-drop numbers are initialized with the values $D_{t,\ell} = 0$ for $t = 0$ or $\ell = 0$. The recursive formula of the egg-drop number

$$D_{t,\ell} = D_{t-1,\ell-1} + D_{t-1,\ell} + 1$$

gives the solution $D_{t,\ell} = \left(\binom{t}{\ell}\right) - 1$. Another connection to the egg-drop numbers appeared in the context of threshold group testing, see [4].

6.6 Programming with Feedback Information on Cell Levels

In this section, we are interested in programming with feedback information on the cell levels. As in Section 6.5, after each programming round it is possible to compare the cell levels with some threshold levels. Here, the goal is to learn about the cells' hardness while programming the cells to their target level. It is very likely in practice that the cells do not have to reach a specific target level but rather reach an interval range that defines a level [3, 13, 18]. The feedback obtained by programming the cells is utilized to have an accurate programming that will eventually place the cells in their target range level.

The problem formulation is as follows. We assume that there is a single cell whose initial level is 0 and whose hardness α satisfies $\alpha \in [\alpha_{\min}, \alpha_{\max}]$ for some known parameters $\alpha_{\min}$ and $\alpha_{\max}$. There are two reference cells $\tau_1 < \tau_2$ and the goal is to program the cell such that its level is in the range $[\tau_1, \tau_2]$. Every algorithm is characterized by its maximum number of programming rounds t and the program voltages $V_1, \ldots, V_t$ on each round, chosen in advance. For $i \in [t]$, ℓ_i denotes the cell's level at the end of the i-th programming round. At the i-th programming round, the cell's level, ℓ_i, is compared with τ_1 and τ_2. If $\ell_i > \tau_2$ then the algorithm fails. If $\tau_1 \leqslant \ell_i \leqslant \tau_2$ then the algorithm terminates successfully. If $\ell_i < \tau_1$ and $i < t$ then the algorithm proceeds to the next round and if $\ell_t < \tau_1$ then the algorithm fails. According to the feedback on the cell's level it is possible to have a better estimate of the cell's hardness and so we denote by $[\alpha_i^{\min}, \alpha_i^{\max}]$ for $i \in [t]$ the range of the cell's hardness at the beginning of the i-th programming round. If the algorithm does not fail then for $i \in [t]$, $\ell_i \leqslant \tau_2$ and if $\ell_i \geqslant \tau_1$ then the algorithm terminates successfully. Therefore, we can not have a better estimate for the lower bound on α and $\alpha_i^{\min} = \alpha_{\min}$ for $i \in [t]$.

Let $t(\tau_1, \tau_2, \alpha_{\min}, \alpha_{\max})$ denote the smallest number of programming rounds of any successful algorithm for all $\alpha \in [\alpha_{\min}, \alpha_{\max}]$. In the next lemma and theorem we show a lower bound on $t(\tau_1, \tau_2, \alpha_{\min}, \alpha_{\max})$.

Lemma 6.6.1. *For any successful algorithm with t programming rounds in the worst case, $\alpha^i_{\max} \geqslant \left(\frac{\tau_1}{\tau_2}\right)^{i-1} \alpha_{\max}$ and $\sum_{j=1}^{i} V_j \leqslant \frac{\tau_2}{\alpha_{\max}} \left(\frac{\tau_2}{\tau_1}\right)^{i-1}$, for $i \in [t]$.*

Proof. We prove this claim by induction on i. On the first round $\alpha_1^{\max} = \alpha_{\max} = \left(\frac{\tau_1}{\tau_2}\right)^{0} \alpha_{\max}$. Let V_1 be the program voltage on the first programming round. Since the algorithm does not fail $\alpha_{\max} V_1 \leqslant \tau_2$ and therefore $V_1 \leqslant \frac{\tau_2}{\alpha_{\max}}$.

Assume the claim is true on the i-th programming round and consider the $(i+1)$-st round for $1 \leqslant i < t$. According to the assumption, $\alpha^i_{\max} \geqslant \left(\frac{\tau_1}{\tau_2}\right)^{i-1} \alpha_{\max}$ and $\sum_{j=1}^{i} V_j \leqslant \frac{\tau_2}{\alpha_{\max}} \left(\frac{\tau_2}{\tau_1}\right)^{i-1}$. If the algorithm proceeds to the $(i+1)$-st round then $\ell_i < \tau_1$ and therefore

$$\alpha^{i+1}_{\max} \cdot \sum_{j=1}^{i} V_j < \tau_1,$$
$$\alpha^{i+1}_{\max} < \frac{\tau_1}{\sum_{j=1}^{i} V_j}.$$

Therefore, the best upper bound on $\alpha^{i+1}_{\max}$ is achieved when $\sum_{j=1}^{i} V_j$ reaches its greatest value, that is,

$$\alpha^{i+1}_{\max} \geqslant \frac{\tau_1}{\frac{\tau_2}{\alpha_{\max}} \left(\frac{\tau_2}{\tau_1}\right)^{i-1}} = \left(\frac{\tau_1}{\tau_2}\right)^{i} \alpha_{\max}.$$

The amount of programmed charge for the next round has to satisfy that the cell's level does not reach level τ_2, in the worst case. Thus,

$$\alpha^{i+1}_{\max} \cdot \sum_{j=1}^{i+1} V_j \leqslant \tau_2,$$
$$\sum_{j=1}^{i+1} V_j \leqslant \frac{\tau_2}{\alpha^{i+1}_{\max}} \leqslant \frac{\tau_2}{\alpha_{\max}} \left(\frac{\tau_2}{\tau_1}\right)^{i}.$$

■

Theorem 6.6.2. *For any* $0 < \tau_1 < \tau_2$, $0 < \alpha_{\min} < \alpha_{\max}$,

$$t(\tau_1, \tau_2, \alpha_{\min}, \alpha_{\max}) \geqslant \left\lceil \frac{\log_2(\alpha_{\max}/\alpha_{\min})}{\log_2(\tau_2/\tau_1)} \right\rceil.$$

Proof. For any successful algorithm, for all $i \in [t]$, $\ell_i \leqslant \tau_2$. Therefore, the worst case for the number of programming rounds is given by the minimum value of i where $\ell_i \geqslant \tau_1$ in the worst case. That is, the minimum value of i such that

$$\alpha_{\min} \cdot \sum_{j=1}^{i} V_j \geqslant \tau_1. \tag{6.2}$$

(Remember that for all i, $\alpha_i^{\min} = \alpha_{\min}$.) According to Lemma 6.6.1, $\sum_{j=1}^{i} V_j \leqslant \frac{\tau_2}{\alpha_{\max}} \left(\frac{\tau_2}{\tau_1}\right)^{i-1}$, and therefore the least positive integer i that satisfies (6.2) has to satisfy also

$$\alpha_{\min} \cdot \frac{\tau_2}{\alpha_{\max}} \left(\frac{\tau_2}{\tau_1}\right)^{i-1} \geqslant \tau_1$$
$$\left(\frac{\tau_2}{\tau_1}\right)^{i} \geqslant \frac{\alpha_{\max}}{\alpha_{\min}},$$
$$i \geqslant \left\lceil \frac{\log_2(\alpha_{\max}/\alpha_{\min})}{\log_2(\tau_2/\tau_1)} \right\rceil.$$

■

Next, we present an algorithm that achieves the above lower bound and hence is optimal. Let $t = \left\lceil \frac{\log_2(\alpha_{\max}/\alpha_{\min})}{\log_2(\tau_2/\tau_1)} \right\rceil$, $V_1 = \frac{\tau_2}{\alpha_{\max}}$ and for $2 \leqslant i \leqslant t$,

$$V_i = \frac{\tau_2}{\alpha_{\max}} \left(\left(\frac{\tau_2}{\tau_1}\right)^{i-1} - \left(\frac{\tau_2}{\tau_1}\right)^{i-2} \right).$$

The algorithm works as follows

1. Program the cell with program voltage V_i.
2. If $i < t$, compare the cell level ℓ_i with the threshold level τ_1. If $\ell_i < \tau_1$ then continue to round $i+1$. Otherwise, stop the algorithm. If $i = t$ the algorithm stops.

The success if this algorithm is proved in the following three lemmas.

Lemma 6.6.3. *For $i \in [t]$, if the algorithm reaches the i-th programming round, then $\alpha_i^{\max} = \left(\frac{\tau_1}{\tau_2}\right)^{i-1} \alpha_{\max}$.*

Proof. We prove this claim by induction on i. On the first programming round we know that $\alpha \in [\alpha_{\min}, \alpha_{\max}]$ and the claim is clear. Assume that the claim is true on the i-th round, where $2 \leqslant i < t$ and $\alpha_i^{\max} = \left(\frac{\tau_1}{\tau_2}\right)^{i-1} \alpha_{\max}$, then

$$V_i = \frac{\tau_2}{\alpha_{\max}} \left(\left(\frac{\tau_2}{\tau_1}\right)^{i-1} - \left(\frac{\tau_2}{\tau_1}\right)^{i-2} \right),$$

and the total programmed voltage is

$$\sum_{j=1}^{i} V_j = \frac{\tau_2}{\alpha_{\max}} + \sum_{j=2}^{i} \frac{\tau_2}{\alpha_{\max}} \left(\left(\frac{\tau_2}{\tau_1}\right)^{j-1} - \left(\frac{\tau_2}{\tau_1}\right)^{j-2} \right) = \frac{\tau_2}{\alpha_{\max}} \left(\frac{\tau_2}{\tau_1}\right)^{i-1}.$$

Therefore, if the algorithm reaches its $(i+1)$-st programming round, then $\ell_i < \tau_1$ and so

$$\alpha \cdot \sum_{j=1}^{i} V_j < \tau_1,$$
$$\alpha < \left(\frac{\tau_1}{\tau_2}\right)^{i} \alpha_{\max}.$$

That is, on the $(i+1)$-st round $\alpha_{i+1}^{\max} = \left(\frac{\tau_1}{\tau_2}\right)^{i} \alpha_{\max}$. ■

Lemma 6.6.4. *For $i \in [t]$, on the i-th programming round,*

$$\ell_i \leqslant \tau_2.$$

Proof. As shown in Lemma 6.6.3, for $i \in [t]$, if the algorithm reaches the i-th programming round then $\sum_{j=1}^{i} V_j = \frac{\tau_2}{\alpha_{\max}} \left(\frac{\tau_2}{\tau_1}\right)^{i-1}$ and $\alpha \leqslant \left(\frac{\tau_1}{\tau_2}\right)^{i-1} \alpha_{\max}$. Therefore, the cell level ℓ_i satisfies

$$\ell_i = \alpha \cdot \sum_{j=1}^{i} V_j = \alpha \cdot \frac{\tau_2}{\alpha_{\max}} \left(\frac{\tau_2}{\tau_1}\right)^{i-1} \leqslant \left(\frac{\tau_1}{\tau_2}\right)^{i-1} \alpha_{\max} \cdot \frac{\tau_2}{\alpha_{\max}} \left(\frac{\tau_2}{\tau_1}\right)^{i-1} = \tau_2.$$

■

Lemma 6.6.5. *At the end of the algorithm the cell level is not less than τ_1.*

Proof. If the algorithm stops on the i-th programming round where $1 \leqslant i < t$ then clearly the cell's level is not less than τ_1. If the algorithm reaches its t-th level, then the total programmed charge is $\sum_{j=1}^{t} V_j = \frac{\tau_2}{\alpha_{\max}} \left(\frac{\tau_2}{\tau_1}\right)^{t-1}$, and the cell's level ℓ_t satisfies

$$\begin{aligned} \ell_t = \alpha \cdot \sum_{j=1}^{t} V_j &\geqslant \alpha_{\min} \cdot \frac{\tau_2}{\alpha_{\max}} \left(\frac{\tau_2}{\tau_1}\right)^{t-1} \\ &= \alpha_{\min} \cdot \frac{\tau_2}{\alpha_{\max}} \left(\frac{\tau_2}{\tau_1}\right)^{\left\lceil \frac{\log_2(\alpha_{\max}/\alpha_{\min})}{\log_2(\tau_2/\tau_1)} \right\rceil - 1} \\ &\geqslant \alpha_{\min} \cdot \frac{\tau_2}{\alpha_{\max}} \cdot \frac{\alpha_{\max}}{\alpha_{\min}} \cdot \frac{\tau_1}{\tau_2} = \tau_1. \end{aligned}$$

■

Corollary 6.6.6. *The above algorithm is optimal. And*

$$t(\tau_1, \tau_2, \alpha_{\min}, \alpha_{\max}) = \left\lceil \frac{\log_2(\alpha_{\max}/\alpha_{\min})}{\log_2(\tau_2/\tau_1)} \right\rceil .$$

6.7 Conclusion

Parallel programming is an important technique for flash memories. And flash memories have a unique iterative and monotonic cell-programming model. In this chapter, we studied parallel programming for flash memories, focusing on its two special properties: shared program voltages and varied programming hardness. We showed algorithms that describe how to implement the parallel programming when the cells' hardness is known. We showed how to modify them in case there is inter-cell interference when cells are programmed. We then proceeded to show how the cells' hardness can be obtained by feedback received while cells are programmed. Finally, we showed how to program a single cell with feedback.

Acknowledgment

This chapter is in part a reprint of the material in the paper: E. Yaakobi, A. Jiang, P.H. Siegel, A. Vardy, and J.K. Wolf, "On The Parallel Programming of Flash Memory Cells," *Proc. IEEE Information Theory Workshop*, Dublin, Ireland, August-September 2010.

Bibliography

[1] A. Berman, Y. Birk, "Error Correction Scheme for Constrained Inter-Cell Interference in Flash Memory", *2nd Annual Non-Volatile Memories Workshop (NVMW'11)*, San Diego, CA, March 2011.

[2] M. Boardman, "The egg-drop numbers," *Mathematics magazine*, vol. 77, no. 5, pp. 368–372, December, 2004.

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

[4] P. Damaschke, "Threshold group testing", *General Theory of Information Transfer and Combinatorics*, LNCS vol. 4123, pp. 707–718, 2006.

[5] G. Dong, S. Li, and T. Zhang, "Using data post-compensation and pre-distortion to tolerate cell-to-cell interference in MLC NAND flash memory," *IEEE Trans. Circuits Sytems I*, vol. 57, no. 10, pp. 2718–2728, October 2010.

[6] D. Haugland, “A bidirectional greedy heuristic for the subspace selection problem,” *Lecture Notes in Computer Science*, vol. 4638, pp. 162-176, August 2007.

[7] A. Jiang and J. Bruck, “On the capacity of flash memories,” in *Proc. Int. Symp. on Information Theory and Its Applications*, pp. 94-99, 2008.

[8] A. Jiang and H. Li, “Optimized cell programming for flash memories,” in *Proc. IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM)*, pp. 914-919, 2009.

[9] Y.J. Kim, H. Son, K.L. Cho, J. Kim, J.J. Kong, and J. Lee, “Modulation Coding for Flash Memories,” to appear in *Proc. IEEE Int. Symp. Inform. Theory*, Saint Petersburg, Russia, July-August 2011.

[10] M. K. Kozlov, S. P. Tarasov, and L. G. HaEijan, “Polynomial solvability of convex quadratic programming,” *Soviet Math. Doklady*, vol. 20, pp. 1108-1111, 1979.

[11] L.A. Lastras-Montaño, M.M. Franceschini, and T. Mittelholzer, “The capacity of the uniform noise rewritable channel with average cost,” *Proc. IEEE Int. Symp. Inform. Theory*, pp. 201–205, Austin, Texas, June 2010.

[12] L.A. Lastras-Montaño, M. Franceschini, T. Mittelholzer, and M. Sharma, “Rewritable storage channels,” *Proc. IEEE Int. Symp. on Inform. Theory and its Applications*, pp. 106–111, Auckland, New Zealand, December 2008.

[13] H. T. Lue, T. H. Hsu, S. Y. Wang, E. K. Lai, K. Y. Hsieh, R. Liu, and C. Y. Lu, “Study of incremental step pulse programming (ISPP) and STI edge effect of BE-SONOS NAND flash,” *Proc. IEEE Int. Symp. on Reliability Physics*, vol. 30, no. 11, pp. 693-694, May 2008.

[14] J.D. Lee, S.H. Hur, and J.D. Choi, “Effects of floating-gate interference on NAND flash memory cell operatio,” *IEEE Electron Device Lett.*, vol. 23, no. 5, pp. 264–266, May 2002.

[15] T. Mittelholzer, M. Franceschini, L.A. Lastras-Montaño, I.M. Elfadel, and M. Sharma, “Rewritable channels with data-dependent noise,” *Proc. IEEE Int. Conf. on Communications*, pp. 1–6, Dresden, Germany, June 2009.

[16] B. K. Natarajan, “Sparse approximate solutions to linear systems,” *SIAM J. Comput.*, vol. 30, no. 2, pp. 227-234, April 1995.

[17] Y. Pan, G. Dong, and T. Zhang, “Exploiting memory device wear-out dynamics to improve NAND flash memory system performance,” *9th USENIX Conf. on File and Storage Technologies*, San Jose, CA, February 2011.

[18] K. D. Suh *et al.*, “A 3.3 V 32 Mb NAND flash memory with incremental step pulse programming,” *IEEE Journal of Solid-State Circuits*, vol. 30, no. 11, pp. 1149-1156, November 1995.

Chapter 7

Storage Coding for Wear Leveling in Flash Memories

7.1 Introduction

Flash memory cells are organized into *blocks*. Each block is further partitioned into multiple *pages*, and every read or write operation accesses a page as a unit. Typically, a page has 2KB to 4KB of data, and 64 to 256 pages comprise a block [5]. Due to the block erasure property of flash memories, every page can be read and written (for the first time) individually, however, rewriting a page (that is, modifying its contents) requires the whole block to be erased and then reprogrammed. Since block erasures degrade the quality of the cells, it is critical to minimize the number of block erasures. It is also critical to balance the number of erasures across different blocks. For this reason, numerous *wear leveling* techniques are widely used in flash-memory systems. The general idea is to balance erasures by migrating data to different locations, especially when data are rewritten [5].

In wear leveling, it is often desirable to move the frequently changing data (so-called hot data) into the same blocks, while storing the mostly static data together in other blocks. Thereby the overall erasures caused by the hot data can be reduced (see [5, 6]). The specific locations to which the data are moved can be optimized not only based on the update frequencies, but also on the correlation among the data. Another important application where data movement is required is *defragmentation of files*. Many file systems (and database systems) implemented in flash take the log-structured approach, wherein updates to files are stored non-consecutively across blocks. This way, wear leveling is achieved and local block erasures are avoided [3]. Consequently, files are frequently fragmented. To improve performance, data have to be moved periodically in order to reorganize the file segments. In database systems or sensors, after bursty incoming data flows are reliably stored, data movement is used to store the data in a categorized manner for efficient analysis. To facilitate data movement, a flash translation layer (FTL) is usually employed to map logical data pages to physical pages in the flash memory [5]. Minimizing the number of block erasures incurred during the data movement process remains a major challenge.

In this chapter, we show that coding techniques can significantly reduce the number block erasures incurred during data movement. In addition to the overall number of erasures, we also consider other parameters, such as coding complexity and extra storage space (number of auxiliary blocks).

7.2 Terms and Concepts

In this section, we formally define the data movement problem, and present some useful concepts.

Definition 7.2.1. (DATA MOVEMENT PROBLEM)
Consider n blocks storing data in a flash memory, and suppose that each block contains m pages. The n blocks are denoted by $B_1, \ldots, B_n$, and the m pages in block B_i are denoted by $p_{i,1}, \ldots, p_{i,m}$ for $i = 1, \ldots, n$. Let $\alpha(i,j)$ and $\beta(i,j)$ be two functions:

$$\alpha(i,j) : \{1,\ldots,n\} \times \{1,\ldots,m\} \rightarrow \{1,\ldots,n\};$$
$$\beta(i,j) : \{1,\ldots,n\} \times \{1,\ldots,m\} \rightarrow \{1,\ldots,m\}.$$

The functions $\alpha(i,j)$ and $\beta(i,j)$ specify the desired data movement. Specifically, the data initially stored in the page $p_{i,j}$ are denoted by $D_{i,j}$, and need to be moved into page $p_{\alpha(i,j),\beta(i,j)}$, for all $(i,j) \in \{1,\ldots,n\} \times \{1,\ldots,m\}$.

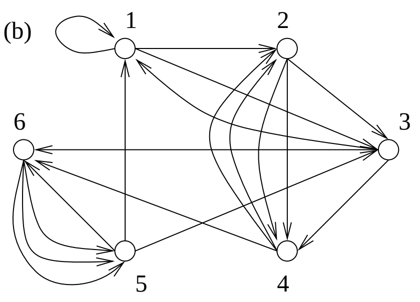
A given number of empty blocks, called auxiliary blocks, can be used in the data movement process, and they need to be erased in the end. To ensure data integrity, at any moment of the data movement process, the data stored in the flash memory blocks should be sufficient for recovering all the original data. The objective is to minimize the total number of block erasures in the data movement process.

Clearly, the functions $\alpha(i,j)$ and $\beta(i,j)$ together have to form a permutation for the mn pages. To avoid trivial cases, we assume that every block has at least one page whose data need to be moved to another block (otherwise, it can be simply excluded from the set of the n

(a)

	i=1	2	3	4	5	6
j=1	3,3	4,1	4,3	2,3	6,3	5,3
2	2,1	4,2	6,1	2,2	3,2	5,2
3	1,3	3,1	1,2	6,2	1,1	5,1

(b)



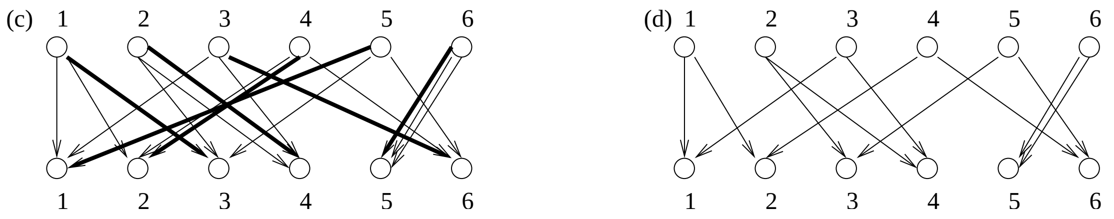


Figure 7.1: Data movement with $n = 6, m = 3$.

blocks considered in the data movement problem). Also note that a block has to be fully erased whenever any of its pages is modified.

Let us now define some terms that are used throughout the chapter. There are two useful graph representations for the data movement problem: the *transition graph* and a *bipartite graph*. In the *transition graph* $G = (V, E)$, $|V| = n$ vertices represent the n data blocks $B_1, \ldots, B_n$. If y pages of data need to be moved from B_i to B_j, then there are y directed edges from B_i to B_j in G. G is a regular directed graph with m outgoing edges and m incoming edges for every vertex. In the *bipartite graph* $H = (V_1 \cup V_2, E')$, V_1 and V_2 each has n vertices that represent the n blocks. If y pages of data are moved from B_i to B_j, there are y directed edges from vertex $B_i \in V_1$ to vertex $B_j \in V_2$. The two graphs are equivalent but are used in different proofs.

Definition 7.2.2.(BLOCK-PERMUTATION SET AND SEMI-CYCLE)
A set of n pages $\{p_{1,j_1}, p_{2,j_2}, \ldots, p_{n,j_n}\}$ that belong to n different blocks is called a block-permutation set if

$$\{\alpha(1, j_1), \alpha(2, j_2), \ldots, \alpha(n, j_n)\} = \{1, 2, \ldots, n\}.$$

If $\{p_{1,j_1}, p_{2,j_2}, \ldots, p_{n,j_n}\}$ is a block-permutation set, then the data they initially store, namely $\{D_{1,j_1}, D_{2,j_2}, \ldots, D_{n,j_n}\}$, are called a block-permutation data set.

Let $z \in \{1, 2, \ldots, n\}$. An ordered set of pages

$$(p_{i_0,j_0}, p_{i_1,j_1}, \ldots, p_{i_{z-1},j_{z-1}})$$

is called a semi-cycle if for $k = 0, 1, \ldots, z-1$, *we have*

$$\alpha(i_k, j_k) = i_{k+1 \bmod z}.$$

Example 7.2.1. The data movement problem shown in Fig. 7.1 exemplifies the construction of the transition graph and the bipartite graph with $n = 6, m = 3$. Subfigure (a) shows the permutation table. The numbers with coordinates (i, j) are $\alpha(i, j), \beta(i, j)$. For example, $(\alpha(1,1), \beta(1,1)) = (3,3)$, and $(\alpha(1,2), \beta(1,2)) = (2,1)$. Subfigure (b) is the transition graph. Subfigure (c) is the bipartite graph representation. The n thick edges are a perfect matching (a block-permutation set). Subfigure (d) is the graph after removing a perfect matching from the bipartite graph. Here for $i = 1, \ldots, n$, vertex i represents block B_i. The $nm = 18$ pages can be partitioned into three block-permutation sets:

$$\{p_{1,1}, p_{2,2}, p_{3,2}, p_{4,2}, p_{5,3}, p_{6,1}\},$$

$$\{p_{1,2}, p_{2,1}, p_{3,3}, p_{4,3}, p_{5,2}, p_{6,2}\},$$

$$\{p_{1,3}, p_{2,3}, p_{3,1}, p_{4,1}, p_{5,1}, p_{6,3}\}.$$

The block-permutation sets can be further decomposed into six semi-cycles:

$$(p_{5,3}, p_{1,1}, p_{3,2}, p_{6,1}), (p_{2,2}, p_{4,2}), (p_{5,2}, p_{3,3}, p_{1,2}, p_{2,1}, p_{4,3}, p_{6,2}),$$

$$(p_{1,3}), (p_{2,3}, p_{3,1}, p_{4,1}), (p_{5,1}, p_{6,3}).$$

Every *semi-cycle* corresponds to a directed cycle in the transition graph, and every *block-permutation set* corresponds to a set of directed cycles that enter and leave every vertex exactly once. It is not a coincidence that the nm pages in the above example can be partitioned into m block-permutation sets. The following theorem shows it holds for the general case.

Theorem 7.2.3. *The* nm *pages can be partitioned into* m *block-permutation sets. Therefore, the* nm *pages of data can be partitioned into* m *block-permutation data sets.*

Proof. The data movement problem can be represented by the *bipartite graph*, where every edge represents a page whose data need to be moved into another block. (See Fig. 7.1 (c) for an example.) It is known that for every bipartite graph $G = (V, E)$ with bipartition $\{A, B\}$ (namely, $A \cap B = \emptyset$ and $A \cup B = V$), we have the Hall's Marriage Theorem [4]:

For $S \subseteq A$*, let* $N(S)$ *denote the set of vertices in the graph* G *that are adjacent to at least one vertex in* S*. (That is, the vertices in* $N(S)$ *are the neighbors of the vertices in* S*.) Then, the graph* G *contains a matching of* A *if and only if* $|N(S)| \geqslant |S|$ *for all* $S \subseteq A$.

For the bipartite graph we are considering here, for $i = 1, \dots, n$, any i vertices in the top layer have im outgoing edges and therefore are connected to at least i vertices in the bottom layer. Therefore, the bipartite graph has a perfect matching. The edges of the perfect matching correspond to a block-permutation set. If we remove those edges, we get a bipartite graph of degree $m - 1$ for every vertex. (See Fig. 7.1 (c), (d).) With the same argument, we can find another perfect matching and reduce the bipartite graph to regular degree $m - 2$. In this way, we partition the nm edges into m block-permutation sets. ■

A perfect matching in the bipartite graph can be found using the Ford-Fulkerson Algorithm [7] for computing maximum flow. The idea is to connect all the n top-layer vertices of the bipartite graph to a source s and connect all the n bottom-layer vertices to a sink t. Then a perfect matching in the bipartite graph is equivalent to a maximum flow of capacity n between the source s and the sink t. The Ford-Fulkerson Algorithm has time complexity $O(n^2m)$, so decomposing the nm edges in the bipartite graph into m perfect matchings has time complexity $O(n^2m^2)$. Therefore, we can partition the nm pages into m block-permutation sets in time $O(n^2m^2)$.

7.3 Coding for Minimizing Auxiliary Blocks

In this work, we focus on the scenario where as few auxiliary blocks as possible are used in the data movement process. In this section, we show that coding techniques can minimize the number of auxiliary blocks. Afterwards, we will study how to use coding to minimize block erasures.

7.3.1 Data Movement without Coding

When coding is not used, data are directly copied from page to page. The following simple example shows that, in the worst case, more than one auxiliary block is needed for data movement. Note that $D_{i,j}$ denotes the data initially stored in the page $p_{i,j}$.

Example 7.3.1. Let $n = m = 2$, and let the functions $\alpha(i,j)$ and $\beta(i,j)$ be:

$$(\alpha(1,1), \beta(1,1)) = (1,1),$$

$$(\alpha(1,2), \beta(1,2)) = (2,2),$$

$$(\alpha(2,1), \beta(2,1)) = (2,1),$$

$$(\alpha(2,2), \beta(2,2)) = (1,2).$$

It is simple to verify that without coding, there is no way to move the data as requested with only one auxiliary block. To see that, assume that only one auxiliary block B_0 is used. Without loss of generality, assume that we first copy the data in B_1 – the data $D_{1,1}$ and $D_{1,2}$ – into B_0, and then erase B_1. In the next step, the only reasonable choice is to write into B_1 the data $D_{1,1}$ and $D_{2,2}$ (which are the data we want to eventually move into B_1). After this writing, B_0 has $D_{1,1}$ and $D_{1,2}$, B_1 has $D_{1,1}$ and $D_{2,2}$, and B_2 has $D_{2,1}$ and $D_{2,2}$. The objective of the data movement has not been met yet. However, we can see that there is no way to proceed: in the next step, if we erase B_0, the data $D_{1,2}$ will be lost; if we erase B_2, the data $D_{2,1}$ will be lost. So the data movement fails. It is simple to verify that no feasible solution exists. Therefore, at least two auxiliary blocks are needed. □

We now show that two auxiliary blocks are sufficient for data movement without coding. The next algorithm uses two auxiliary blocks, which are denoted by B_0 and B_0'. It operates in a way similar to bubble sort. And it sorts the data of the m block-permutation data sets in parallel.

Algorithm 7.3.1. (BUBBLE-SORT-BASED DATA MOVEMENT)

```
Decompose the $nm$ pages of data into $m$ block-permutation
data sets.
For $i = 1, \ldots, n-1$
{
    For $j = i+1, \ldots, n$
    {
        Copy all the data of $B_i$ into $B_0$;
        Copy all the data of $B_j$ into $B_0'$;
        Erase $B_i$ and $B_j$;
        For $k = 1, \ldots, m$
        {
            Let $D_{i_1,j_1}$ and $D_{i_2,j_2}$ be the two pages of data in $B_0$ and $B_0'$,
            respectively, that belong to the $k$-th block-permutation data set.
            Let $p_{i,j_3}$ be the unique page in $B_i$ such that when the data move-
            ment process ends, the data stored in $p_{i,j_3}$ will be from the $k$-th
            block-permutation data set.
```

```
                If $\alpha(i_2, j_2) = i$ (which implies $\beta(i_2, j_2) = j_3$ and $\alpha(i_1, j_1) \neq$
                $i$), copy the data $D_{i_2,j_2}$ into the page $p_{i,j_3}$; otherwise, copy the
                data $D_{i_1,j_1}$ into the page $p_{i,j_3}$.
            }
            Write into $B_j$ the $m$ pages of data that are in $B_0$ or $B'_0$ but not in $B_i$.
            Erase $B_0$ and $B'_0$.
        }
    }
```

In the above algorithm, for every block-permutation data set, its data are not only sorted in parallel with other block-permutation data sets, but are also always dispersed in n blocks (with every block holding one page of its data). The algorithm uses $O(n^2)$ erasures. (The n blocks $B_1, \ldots, B_n$ are each erased $n-1$ times, while the two auxiliary blocks B_0 and B'_0 are each erased $\binom{n}{2}$ times.) If instead of bubble sorting, we use more efficient sorting networks such as the Batcher sorting network [2] or the AKS network [1], the number of erasures can be further reduced to $O(n \log^2 n)$ and $O(n \log n)$, respectively. For simplicity we skip the details.

7.3.2 Storage Coding with One Auxiliary Block

In Algorithm 7.3.1, the only function of the two auxiliary blocks B_0 and B'_0 is to store the data of the data blocks B_i, B_j when the data in B_i, B_j are being swapped. We now show how coding can help reduce the number of auxiliary blocks to one, which is clearly the best possible. Let B_0 denote the only auxiliary block, and let $p_{0,1}, p_{0,2}, \ldots, p_{0,m}$ denote its pages. For $k = 1, \ldots, m$, statically store in page $p_{0,k}$ the bit-wise exclusive-OR of the n pages of data in the k-th block-permutation data set. We make such a change in Algorithm 7.3.1:

When the data in B_i, B_j are swapped, instead of erasing them together, we first erase B_i and write data into B_i, then erase B_j and write data into B_j.

This is feasible because for every block-permutation data set, there are always at least n pages of data related to it stored in the $n+1$ blocks: $n-1$ pages of those data are the original data in the block-permutation data set, and the other page of data are the bit-wise exclusive-OR of the data of the block-permutation data set. The total number of block erasures here is of the same order as the algorithm without coding. Therefore, if the AKS network is used for swapping the data, $O(n \log n)$ block erasures will be used in total.

7.4 Efficient Storage Coding over GF(2)

In this section, we present a data movement algorithm that uses only one auxiliary block and $2n$ erasures. It erases every block either once or twice, which is well balanced. The algorithm uses coding over GF(2) and is very efficient.

For convenience, let us assume for now that every block has only one page. The results will be naturally extended to the general case where every block has m pages. (Note that the erasure of a block will affect all the m block-permutation data sets. So when $m \geqslant 2$, the sequence of block erasures need to be compatible for the data movement of all those m sets.) Let B_0 denote the auxiliary block, and let p_0 denote its page. For $i = 1, \dots, n$, let p_i denote the page in the block B_i, and let D_i denote the data initially stored in the page p_i. Let

$$\alpha : \{1, \dots, n\} \to \{1, \dots, n\}$$

be the permutation such that the data D_i need to be moved into the page $p_{\alpha(i)}$. Let α^{-1} be the inverse permutation of α. Say that the n pages $p_1, p_2, \dots, p_n$ can be partitioned into t semi-cycles, denoted by

$$C_1, C_2, \dots, C_t.$$

Note that since right now we consider a block to have only one page, a semi-cycle is just a cycle in the permutation α. Every semi-cycle C_i $(1 \leqslant i \leqslant t)$ has a special page called *tail*, defined as follows: if p_j is the *tail* of C_i, then for every other page $p_k \in C_i$, we have $j > k$.

We use "$\oplus$" to represent the bit-wise exclusive-OR of data. The following algorithm uses $2n$ block erasures to move data. It consists of two passes: the *forward pass* and the *backward pass*. Note that in the algorithm below, whenever some data are about to be written into a page, the data can be efficiently computed from the existing data in the flash memory blocks (namely, from the data currently stored in the flash memory). The details will be clear later. Also note that for $i = 1, 2, \dots, n$, $D_{\alpha^{-1}(i)}$ is the data that need to be moved into the page p_i.

Algorithm 7.4.1. (GF(2)-CODING-BASED DATA MOVEMENT)

```
FORWARD PASS:
For $i = 1, 2, \dots, n$ do:
        If $p_i$ is not the tail of its semi-cycle, write
                $D_i \oplus D_{\alpha^{-1}(i)}$
        into the page $p_{i-1}$; otherwise, write
                $D_i$
        into the page $p_{i-1}$. Then, erase the block $B_i$.
BACKWARD PASS:
For $i = n, n-1, \ldots, 1$ do:
        Write
                $D_{\alpha^{-1}(i)}$
        into the page $p_i$. Then, erase the block $B_{i-1}$.
```

Example 7.4.1. Figure 7.2 gives an example of the execution of Algorithm 7.4.1 with $n = 8$ and $t = 2$. Here

$$(\alpha(1), \alpha(2), \ldots, \alpha(8)) = (3, 6, 8, 1, 2, 5, 4, 7).$$

Consequently, we have

$$(\alpha^{-1}(1), \alpha^{-1}(2), \ldots, \alpha^{-1}(8)) = (4, 5, 1, 7, 6, 2, 8, 3).$$

The two semi-cycles are $(p_1, p_3, p_8, p_7, p_4)$ and (p_2, p_6, p_5). In Figure 7.2, each row is a step of Algorithm 7.4.1. The numbers are the data in the blocks. (For convenience, we use i to denote data D_i in the figure for $i = 1, 2, \ldots, 8$.) The rightmost column describes the computation performed for this step, where δ_i denotes the data in p_i then. □

The correctness of Algorithm 7.4.1 depends on whether the data written into a page can always be derived from the existing data in the flash memory blocks. Theorem 7.4.2 shows this is true.

Theorem 7.4.2. *When Algorithm 7.4.1 is running, at any moment, for $i = 1, 2, \ldots, n$, if the data D_i are not stored in the $n+1$ blocks $B_0, B_1, \ldots, B_n$, then there must exist a set of data*

$$\{D_i \oplus D_{j_1}, D_{j_1} \oplus D_{j_2}, D_{j_2} \oplus D_{j_3}, \ldots, D_{j_{k-1}} \oplus D_k, D_k\}$$

that are all stored in the $n+1$ blocks. Therefore, D_i can be easily obtained by computing the bit-wise exclusive-OR of the data in the set.

B_0	B_1	B_2	B_3	B_4	B_5	B_6	B_7	B_8	Operation
forward pass									
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	$\delta_1 \oplus \delta_4$
$1 \oplus 4$		2	3	4	5	6	7	8	$\delta_2 \oplus \delta_5$
$1 \oplus 4$	$2 \oplus 5$		3	4	5	6	7	8	$\delta_3 \oplus \delta_0 \oplus \delta_4$
$1 \oplus 4$	$2 \oplus 5$	$3 \oplus 1$		4	5	6	7	8	$\delta_4 \oplus \delta_7$
$1 \oplus 4$	$2 \oplus 5$	$3 \oplus 1$	$4 \oplus 7$		5	6	7	8	$\delta_5 \oplus \delta_6$
$1 \oplus 4$	$2 \oplus 5$	$3 \oplus 1$	$4 \oplus 7$	$5 \oplus 6$		6	7	8	copy δ_6
$1 \oplus 4$	$2 \oplus 5$	$3 \oplus 1$	$4 \oplus 7$	$5 \oplus 6$	6		7	8	$\delta_7 \oplus \delta_8$
$1 \oplus 4$	$2 \oplus 5$	$3 \oplus 1$	$4 \oplus 7$	$5 \oplus 6$	6	$7 \oplus 8$		8	copy δ_8
$1 \oplus 4$	$2 \oplus 5$	$3 \oplus 1$	$4 \oplus 7$	$5 \oplus 6$	6	$7 \oplus 8$	8		
backward pass									
$1 \oplus 4$	$2 \oplus 5$	$3 \oplus 1$	$4 \oplus 7$	$5 \oplus 6$	6	$7 \oplus 8$	8		$\delta_7 \oplus \delta_6 \oplus \delta_3 \oplus \delta_0 \oplus \delta_2$
$1 \oplus 4$	$2 \oplus 5$	$3 \oplus 1$	$4 \oplus 7$	$5 \oplus 6$	6	$7 \oplus 8$		3	$\delta_6 \oplus \delta_3 \oplus \delta_0 \oplus \delta_2 \oplus \delta_8$
$1 \oplus 4$	$2 \oplus 5$	$3 \oplus 1$	$4 \oplus 7$	$5 \oplus 6$	6		8	3	$\delta_5 \oplus \delta_4 \oplus \delta_1$
$1 \oplus 4$	$2 \oplus 5$	$3 \oplus 1$	$4 \oplus 7$	$5 \oplus 6$		2	8	3	$\delta_4 \oplus \delta_1 \oplus \delta_6$
$1 \oplus 4$	$2 \oplus 5$	$3 \oplus 1$	$4 \oplus 7$		6	2	8	3	$\delta_3 \oplus \delta_0 \oplus \delta_2 \oplus \delta_8$
$1 \oplus 4$	$2 \oplus 5$	$3 \oplus 1$		7	6	2	8	3	$\delta_2 \oplus \delta_8$
$1 \oplus 4$	$2 \oplus 5$		1	7	6	2	8	3	$\delta_1 \oplus \delta_6$
$1 \oplus 4$		5	1	7	6	2	8	3	$\delta_0 \oplus \delta_3$
	4	**5**	**1**	**7**	**6**	**2**	**8**	**3**	

Figure 7.2: Example of Algorithm 7.4.1. δ_i denotes the data in page p_i, for $i = 0, 1, \ldots, 8$.

Proof. Consider a semi-cycle C_i ($1 \leqslant i \leqslant t$), and let us denote its pages by

$$p_{i_1}, p_{i_2}, \ldots, p_{i_x}.$$

Without loss of generality (WLOG), assume

$$\alpha(i_j) = i_{j+1}$$

for $j = 1, 2, \ldots, x-1$ and

$$\alpha(i_x) = i_1.$$

Also assume that p_{i_1} is the "tail" of the semi-cycle, namely, $i_1 > i_j$ for $j = 2, 3, \ldots, x$. Now imagine a directed path S as follows:

1. S has x vertices, representing the data D_{i_1}, D_{i_2}, …, D_{i_x};

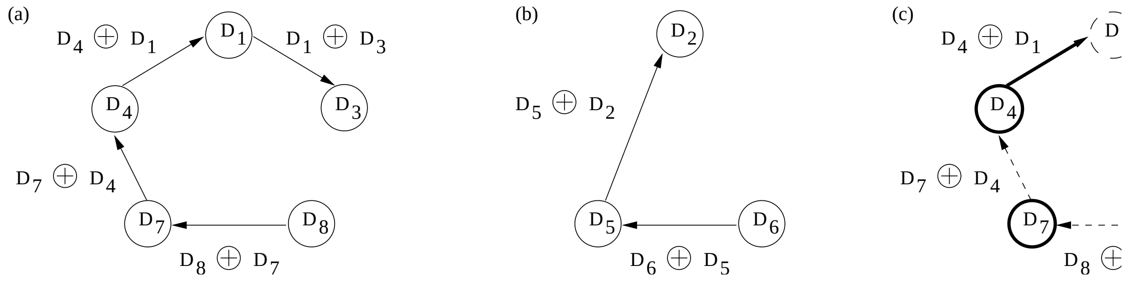


Figure 7.3: The directed path S of a semi-cycle, whose vertices and edges represent data.

2. There is a directed edge from D_{i_j} to $D_{i_{j+1}}$ for $j = 1, 2, \ldots, x-1$. The edge represents the data
$$D_{i_j} \oplus D_{i_{j+1}}.$$

For example, the data movement problem in Example 7.4.1 has two semi-cycles, (p_2, p_6, p_5) and $(p_1, p_3, p_8, p_7, p_4)$. We show the corresponding directed path S in Fig. 7.3. Subfigure (a) is the directed path S for semi-cycle $(p_1, p_3, p_8, p_7, p_4)$, and subfigure (b) is the directed path S for semi-cycle (p_2, p_6, p_5).

The directed path S corresponding to a semi-cycle, whose vertices and edges represent data. (a) The directed path S for semi-cycle $(p_1, p_3, p_8, p_7, p_4)$. (b) The directed path S for semi-cycle (p_2, p_6, p_5). (c) The stored and un-stored data after three block erasures in the "forward-pass" of the data-movement algorithm. The vertices and edges of solid thick lines represent the data that are stored at that moment. The vertices and edges of dashed thin lines represent the data that are not stored at that moment.

Consider the *forward pass* in the algorithm. In this pass, for $j = 2, 3, \ldots, x$, right before the data D_{i_j} are erased, the data $D_{i_{j-1}} \oplus D_{i_j}$ are stored. Note that D_{i_j} corresponds to a vertex in the directed path S, and $D_{i_{j-1}} \oplus D_{i_j}$ corresponds to the directed edge entering that vertex in S. So for every vertex in S whose corresponding data have been erased, there is a directed sub-path in S entering it with this property: "the data represented by the edges in this sub-path, as well as the data represented by the starting vertex of the sub-path, are all stored in the $n+1$ blocks." This is the same as the condition stated in the theorem. (For instance, for the data movement problem in Example 7.4.1, after three block erasures, the stored and un-stored data after three block erasures in the "forward-pass" of the data-movement algorithm are as shown in Fig. 7.3 (c). The vertices and edges of solid thick lines represent the data that are stored at that moment. The vertices and edges of dashed thin lines represent the data that are not stored at that moment.

As an example, consider the erased data D_3. The corresponding sub-path entering it contains the data D_4, $D_4 \oplus D_1$ and $D_1 \oplus D_3$, which are stored and can be used to recover D_3.)

When the *forward pass* of the algorithm ends, the data represented by the vertex D_{i_1} and all the edges in S are all stored in the $n+1$ blocks. Clearly, all the original data can be recovered.

Now consider the *backward pass* in the algorithm. In this pass, first, the data D_{i_x} are stored and then the data D_{i_1} are erased. Then, for $j = 1, 2, \ldots, x-1$, right before the data $D_{i_j} \oplus D_{i_{j+1}}$ are erased, the data D_{i_j} are stored. Note that D_{i_j} corresponds to a vertex in the directed path S, and $D_{i_j} \oplus D_{i_{j+1}}$ corresponds to the directed edge leaving that vertex in S. So for every vertex in S whose corresponding data have been erased, there is a directed sub-path in S leaving it with this property: "the data represented by the edges in this sub-path, as well as the data represented by the end vertex of the sub-path, are all stored in the $n+1$ blocks." This is the same as the condition stated in the theorem. So the conclusion holds. ■

Algorithm 7.4.1 can be easily extended to the general case where every block has $m \geqslant 1$ pages. Use the algorithm to process the m block-permutation data sets in parallel, in the same way as Algorithm 7.3.1. Specifically, for $i = 1, \ldots, n$ and $j = 1, \ldots, m$, let $p_{i,k(i,j)}$ denote the unique page in B_i such that some data in the j-th block-permutation data set need to be moved into $p_{i,k(i,j)}$. In the algorithm, every time B_i is erased, write the data related to the j-th block-permutation data set into $p_{i,k(i,j)}$. Since every block-permutation set occupies exactly one page in each block, there will be no conflict in writing.

7.5 Storage Coding with Minimized Number of Erasures

In this section, we present an algorithm that uses at most $2n-1$ erasures, which is worst-case optimal. It erases every block either once or twice, which is well balanced. We further show that it is NP hard to minimize the number of erasures for every given instance, but our algorithm provides a 2-approximation. Namely, it uses at most twice the number of block erasures compared to the optimal solution.

7.5.1 Optimal Solution with Canonical Labelling

The n blocks initially storing data can be labelled by $B_1, \ldots, B_n$ in $n!$ different ways. Let y be an integer in $\{0, 1, \ldots, n-2\}$. We call a labelling of the n blocks that satisfies the following constraint a *canonical labelling with parameter* y:

"For $i = y+1, y+2, \ldots, n-2$ and $j = i+2, i+3, \ldots, n$, no data initially stored in the block B_j need to be moved into the block B_i."

Trivially, any labelling is a canonical labelling with parameter $n-2$. However, given an instance of the data movement problem, it is difficult to find a canonical labelling that minimizes the value of y.

We now present a data-movement algorithm for blocks that have a canonical labelling with parameter y. It uses one auxiliary block B_0, and uses

$$n + y + 1 \leqslant 2n - 1$$

erasures. So the smaller y is, the better. For convenience, let us again assume that every block contains only one page, and let $p_i, D_i, \alpha, \alpha^{-1}$ be as defined in the previous section. Let r denote the number of bits in a page. [1] The algorithm can be naturally extended to the general case, where every block has $m \geqslant 1$ pages, in the same way as introduced in the previous section.

Algorithm 7.5.1. (DATA MOVEMENT WITH LINEAR CODING)
This algorithm is for blocks that have a canonical labelling with parameter $y \in \{0, 1, \ldots, n-2\}$. *Let* $\gamma_1, \gamma_2, \ldots, \gamma_n$ *be distinct non-zero elements in the field* $\mathrm{GF}(2^r)$.

STEP 1: *For* $i = 0, 1, \ldots, y$ *do: Erase* B_i *(for* $i = 0$ *there is no need to erase* B_0*), and write into* p_i *the data* $\sum_{k=1}^{n} \gamma_k^i D_k$.

STEP 2: *For* $i = y+1, y+2, \ldots, n$ *do: Erase* B_i*, and write into* p_i *the data* $D_{\alpha^{-1}(i)}$.

STEP 3: *For* $i = y, y-1, \ldots, 1$ *do: Erase* B_i*, and write into the page* p_i *the data* $D_{\alpha^{-1}(i)}$. *Finally, erase* B_0.

Theorem 7.5.2. *Algorithm* 7.5.1 *is correct and uses*

$$n + y + 1 \leqslant 2n - 1$$

erasures. (Note that the algorithm assumes that the blocks have a canonical labelling with parameter y*.)*

Proof. We show that each time a block B_i is erased, it is feasible to generate all the n pages of original data using the current data stored in the other n pages. Denote by δ_i, $0 \leqslant i \leqslant n$, the

[1]When r is greater than what is needed by Algorithm 7.5.1, which is nearly always true in practice, we can partition each page into bit strings of an appropriate length, and apply the algorithm to the strings in parallel.

current data stored in the page p_i, which are a linear combination of the n pages of original data. The linear combination written in each page can be represented by a matrix multiplication

$$H \cdot (D_1, D_2, \ldots, D_n)^T = (\delta_0, \ldots, \delta_{i-1}, \delta_{i+1}, \ldots, \delta_n)^T.$$

The matrix H defines the linear combination of the original data written into each page. Consider the first step of the algorithm when the block B_i is erased. The data written in p_h, for $0 \leqslant h \leqslant i-1$, are

$$\delta_h = \sum_{k=1}^{n} \gamma_k^h D_k,$$

and the data stored in p_h, for $i+1 \leqslant h \leqslant n$, are

$$\delta_h = D_h.$$

The matrix representation of this problem is

$$\left(\begin{array}{cccc} 1 & 1 & \cdots & 1 \\ \gamma_1 & \gamma_2 & \cdots & \gamma_n \\ \gamma_1^2 & \gamma_2^2 & \cdots & \gamma_n^2 \\ \vdots & \vdots & \ddots & \vdots \\ \gamma_1^{i-1} & \gamma_2^{i-1} & \cdots & \gamma_n^{i-1} \\ \hline \multicolumn{2}{c}{0_{(n-i)\times i}} & \multicolumn{2}{c}{I_{n-i}} \end{array}\right) \cdot \begin{pmatrix} D_1 \\ D_2 \\ D_3 \\ \vdots \\ D_{n-1} \\ D_n \end{pmatrix} = \begin{pmatrix} \delta_0 \\ \vdots \\ \delta_{i-1} \\ \delta_{i+1} \\ \vdots \\ \delta_n \end{pmatrix}$$

where $0_{(n-i)\times i}$ is the zero matrix of size $(n-i)\times i$, and I_{n-i} is the unit matrix of size $(n-i)\times(n-i)$. Since this matrix is invertible, it is feasible to generate all the original data and in particular, the required data that need to be written into p_i.

For $i = y+1, y+2, \ldots, n$, after erasing the block B_i during the second step of the algorithm, the data stored in p_h, for $0 \leqslant h \leqslant y$, are $\delta_h = \sum_{k=1}^{n} \gamma_k^h D_k$. The data written into p_h, for $y+1 \leqslant h \leqslant i-1$, are $\delta_h = D_{\alpha^{-1}(h)}$, and the data stored in p_h, for $i+1 \leqslant h \leqslant n$, are $\delta_h = D_h$. These equations are represented as follows:

$$\left(\begin{array}{cccc} 1 & 1 & \cdots & 1 \\ \gamma_1 & \gamma_2 & \cdots & \gamma_n \\ \gamma_1^2 & \gamma_2^2 & \cdots & \gamma_n^2 \\ \vdots & \vdots & \ddots & \vdots \\ \gamma_1^{y} & \gamma_2^{y} & \cdots & \gamma_n^{y} \\ \hline \multicolumn{4}{c}{A_{n-i}} \end{array}\right) \cdot \begin{pmatrix} D_1 \\ D_2 \\ D_3 \\ \vdots \\ D_{n-1} \\ D_n \end{pmatrix} = \begin{pmatrix} \delta_0 \\ \vdots \\ \delta_{i-1} \\ \delta_{i+1} \\ \vdots \\ \delta_n \end{pmatrix},$$

where A_{n-i} is a matrix of size $(n-y-1)\times n$ defined as follows:

1. The h-th row of the matrix A_{n-i} for $1 \leqslant h \leqslant i-y-1$ is a unit vector of length n containing a one in its $(\alpha^{-1}(y+h))$-th entry.
2. The h-th row of the matrix A_{n-i} for $i-y \leqslant h \leqslant n-y-1$ is a unit vector that contains a one in its $(y+h+1)$-th entry.

Since there are no data that are moved from block B_j to block B_i, where $y+1 \leqslant i \leqslant n-2$ and $i+2 \leqslant j \leqslant n$, the first $i-y-1$ row vectors of the matrix A_{n-i} are different from the last $n-i$ row vectors of the matrix A_{n-i}. Therefore, the matrix A_{n-i} contains a set of unit vectors where all the vectors are different from each other. If we calculate the determinant of the matrix on the left hand side according to the rows of the matrix A_{n-i}, then we are left with an $(y+1) \times (y+1)$ matrix of the form:

$$\begin{pmatrix} 1 & 1 & 1 & \cdots & 1 & 1 \\ \gamma_{i_1} & \gamma_{i_2} & \gamma_{i_3} & \cdots & \gamma_{i_y} & \gamma_{i_{y+1}} \\ \gamma_{i_1}^2 & \gamma_{i_2}^2 & \gamma_{i_3}^2 & \cdots & \gamma_{i_y}^2 & \gamma_{i_{y+1}}^2 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ \gamma_{i_1}^y & \gamma_{i_2}^y & \gamma_{i_3}^y & \cdots & \gamma_{i_y}^y & \gamma_{i_{y+1}}^y \end{pmatrix}$$

and its determinant is not zero because it is a Vandermonde matrix. Therefore, the matrix on the left hand side is invertible, and it is feasible to generate all the original data D_i, $1 \leqslant i \leqslant n$, and in particular the data $D_{\alpha^{-1}(i)}$ that need to be written into the page p_i.

For $i = y, y-1, \ldots, 1$, after erasing the block B_i during the third step of the algorithm, the data stored in p_h, for $0 \leqslant h \leqslant i-1$, are $\delta_h = \sum_{k=1}^{n} \gamma_k^h D_k$, and the data stored in p_h, for $i+1 \leqslant h \leqslant n$, are $\delta_h = D_{\alpha^{-1}(h)}$. Therefore, the matrix representing this equations is

$$\left(\begin{array}{cccc} 1 & 1 & \cdots & 1 \\ \gamma_1 & \gamma_2 & \cdots & \gamma_n \\ \gamma_1^2 & \gamma_2^2 & \cdots & \gamma_n^2 \\ \vdots & \vdots & \ddots & \vdots \\ \gamma_1^{i-1} & \gamma_2^{i-1} & \cdots & \gamma_n^{i-1} \\ \hline \multicolumn{4}{c}{P_{n-i}} \end{array}\right) \cdot \begin{pmatrix} D_1 \\ D_2 \\ D_3 \\ \vdots \\ D_{n-1} \\ D_n \end{pmatrix} = \begin{pmatrix} \delta_0 \\ \vdots \\ \delta_{i-1} \\ \delta_{i+1} \\ \vdots \\ \delta_n \end{pmatrix},$$

where P_{n-i} is a matrix consisting of $n-i$ row vectors of length n, and its h-th row vector, $1 \leqslant h \leqslant n-i$, is a unit vector of length n which has a one in its $\alpha^{-1}(i+h)$-th entry and zero elsewhere. As before, all the unit vectors in the matrix P_{n-i} are different from each other. Therefore the matrix on the left hand side is invertible, and it is feasible to generate all the original data D_i, $1 \leqslant i \leqslant n$, and in particular the data $D_{\alpha^{-1}(i)}$ that need to be written into the page p_i. ■

The above algorithm uses Reed-Solomon codes for data movement. It can be extended to general MDS codes.

The following theorem shows an interesting property of canonical labelling. Note that since every block has some data that need to be moved into it from some other block, every block needs to be erased at least once. So at least $n + 1$ erasures (including erasing the auxiliary block) are needed in any case.

Theorem 7.5.3. *Assume r, the number of bits in a page, is sufficiently large, and let $y \in \{0, 1, \ldots, n-2\}$. There is a data-movement solution using*

$$n + y + 1$$

erasures if and only if there is a canonical labelling of the blocks with parameter y.

Proof. First, assume that there is a data-movement solution using $n + y + 1$ erasures. Since every block (including the auxiliary block) is erased at least once, there are at least $n - y$ blocks that are erased only once in the solution. Pick $n - y$ blocks erased only once and label them as $B_{y+1}, B_{y+2}, \ldots, B_n$ this way: "in the solution, when $y + 1 \leqslant i < j \leqslant n$, B_i is erased before B_j." Label the other y blocks as $B_1, \ldots, B_y$ arbitrarily. Let us use contradiction to prove that no data in B_j need to be moved into B_i, where $i \geqslant y + 1$, $j \geqslant i + 2$.

Assume some data in B_j need to be moved into B_i. After B_i is erased, those data must be written into B_i because B_i is erased only once. When the solution erases B_{i+1} (which happens before B_j is erased), the data mentioned above exist in both B_i and B_j. So at this moment, there are at most $nm - 1$ pages of distinct data; however, it is impossible to recover all the nm pages of original data using only $nm - 1$ pages of distinct data. So there is a contradiction. Therefore, with the above labelling, we have already found a canonical labelling with parameter y. The other direction of the proof comes from the existence of Algorithm 7.5.1. ■

We can easily make Algorithm 7.5.1 use $2n - 1$ erasures by letting $y = n - 2$ and using an arbitrary block labelling. On the other hand, $2n - 1$ erasures are necessary in the worst case. To see that, consider an instance where $m \geqslant n$ and every block has some data that need to be moved into every other block. For such an instance, a canonical labelling has to have $y = n - 2$, which implies $n + y + 1 = 2n - 1$ erasures by Theorem 7.5.3. So Algorithm 7.5.1 is worst-case optimal.

7.6 Conclusion and Future Research

In this chapter, we studied the data movement problem for flash memories. We presented sorting-based algorithms that do not utilize coding, which can use as few as $O(n \log n)$ erasures for moving data among n blocks. We showed that coding techniques can not only minimize the number of auxiliary blocks, but also reduce the number of erasures to $O(n)$. In particular, we presented a solution based on coding over $\mathrm{GF}(2)$ that requires only $2n$ erasures. We further presented a linear-coding solution that requires at most $2n - 1$ erasures, which is worst-case optimal. Both solutions based on coding achieve an approximation ratio of two with respect to the minimum possible number of block erasures for each instance. They also balance the number of erasures in different blocks very well.

The data movement problem studied here can have numerous practical variations. In one variation, the data to be moved into each block are specified, but the order of the data in that block is allowed to be arbitrary. The algorithms presented in this work can easily solve this variation of the problem by first assigning an arbitrary page order to each block (which does not affect the performance of the algorithms). In another variation, we are only given a specification as to which group of data needs to be moved into the same block, without specifying which block. Furthermore, the final data may be a function of the data originally stored in the blocks. Such variations require new solutions for optimal performance. They remain open for future research.

Acknowledgment

This chapter is in part a reprint of the material in the paper: A. Jiang, R. Mateescu, E. Yaakobi, J. Bruck, P.H. Siegel, A. Vardy, and J.K. Wolf, “Storage Coding for Wear Leveling in Flash Memories,” *IEEE Transactions on Information Theory*, vol. 56, no. 10, pp. 5290–5299, October 2010.

Bibliography

[1] M. Ajtai, J. Komlós and E. Szemerédi, “An $O(n \log n)$ sorting network,” in *Proc. Fifteenth Annual ACM Symposium on Theory of Computing (STOC)*, Boston, Massachusetts, USA, April 25-27, 1983, pp. 1–9.

[2] K. E. Batcher, “Sorting networks and their applications,” in *Proc. the AFIPS Spring Joint*

Computer Conference, Atlantic City, New Jersey, USA, April 30 - May 2, 1968, pp. 307–314.

[3] H. Dai, M. Neufeld and R. Han, "ELF: An Efficient Log-structured Flash file system for micro sensor nodes," in *Proc. 2nd International Conference on Embedded Networked Sensor Systems (SenSys)*, Baltimore, Maryland, USA, November 3-5, 2004, pp. 176–187.

[4] R. Diestel, *Graph Theory*, 2nd edition, Chapter 2, Springer, 2000.

[5] E. Gal and S. Toledo, "Algorithms and data structures for flash memories," in *ACM Computing Surveys*, vol. 37, pp. 138-163, June 2005.

[6] J. Hsieh, T. Kuo and L. Chang, "Efficient identification of hot data for flash memory storage systems," in *ACM Transactions on Storage*, vol. 2, no. 1, pp. 22-40, 2006.

[7] J. Kleinberg and E. Tardos, *Algorithm Design*, 1st edition, Chapter 7, Pearson Education Inc., 2005.

Chapter 8

Error Characterization and Coding Schemes for Flash Memories

8.1 Introduction

Flash memory chips may use single-level cell (SLC) technology, where each cell can store one binary digit, or multi-level cell (MLC) technology, where each cell can store multiple binary digits. In this work, we assume that MLC chips store two bits in a cell. First generation flash storage devices have used only low-redundancy codes that offer minimal error correction and detection capabilities, such as single-bit error-correcting Hamming codes and error-detecting cyclic redundancy check (CRC) codes. The demand for increased storage capacity, coupled with the introduction of MLC flash technology, has created the need for more powerful ECC methods, such as BCH codes and Low-Density Parity-Check (LDPC) codes.

The design of effective error-correcting codes requires a comprehensive understanding of the error mechanisms and error characteristics of flash memories. To help address this need, we used in this work an extensive empirical database of errors observed during erase, write, and read operations on a flash memory device. Error statistics were gathered from several blocks on SLC and MLC flash memory chips. For each block, we repeated continuously the following process hundreds of thousands to millions of times:

1. Erase the block.

2. Write pseudo-random data into the block.

3. Read the block and identify errors by comparing the originally recorded data to the data

that was read.

Using this database of errors, we analyzed the error behavior on a block-, page-, and bit-level and classified the observed error types. This information was used in order to suggest a new scheme of error-correcting codes for MLC flash. Furthermore, we explored the implementation of WOM-codes, which were extensively studied in Chapters 3 and 4, for SLC flash.

Remark 8.1.1. We note that the experiments were conducted in a controlled laboratory environment, and the results do not reflect the impact of other performance-related factors such as varying time intervals between erasures, ambient temperature changes, and multiple read operations between erasures. Consequently, the observed lifetimes of the tested flash blocks were much longer than the lifetimes specified by the manufacturer. Also, it should be pointed out that we collected the error data from only a few blocks on each chip, so our results and conclusions do not account for possible variability among blocks on any given chip. A similar disclaimer applies to flash devices produced by different manufacturers. Therefore, we do not pretend to give in this chapter a complete and comprehensive study of error characteristics in flash memories. For further discussion of flash memory characterization, see [3].

8.2 Flash Memory Structure

A flash memory chip is built from floating-gate cells which are organized in blocks. Each block typically contains either 64 pages (SLC) or 128 pages (MLC), where the size of a page can range from 2KB to 8KB [1].

In SLC flash, each cell has two levels and stores one bit. A non-programmed cell represents bit value ‘1’ and once it is charged the bit value is ‘0’ (see Fig. 8.1). In MLC flash, each cell has four levels and stores two bits. The left bit among the two bits is called the Most Significant Bit (MSB) and the right bit is the Least Significant Bit (LSB). The cell has four levels and the mapping between charge values and bit values is depicted in Fig. 8.1.

A typical SLC block consists of 32 rows of 2^{15} cells, such that each row contains two pages. One page consists of the first 2^{14} cells in each row and another page consists of the last 2^{14} cells in the row. A typical layout of the pages within an SLC block is demonstrated in Table 8.1. In MLC flash, the two bits within a single cell are not mapped to the same page. Rather, the collection of MSB’s from a group of cells constitute a page called the MSB page and, similarly, the LSB’s from the same group of cells form a page called the LSB page. The layout of an MLC block is similar to that of an SLC block, as depicted in Table 8.2.

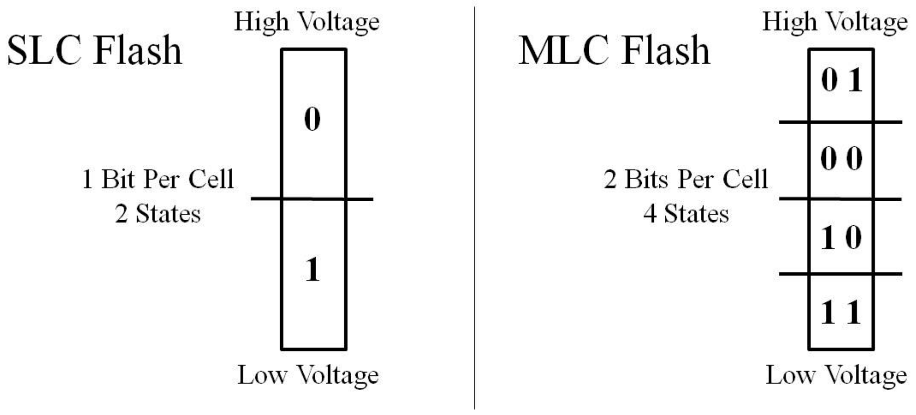


Figure 8.1: Mappings of cell levels to binary representations in SLC and MLC flash.

Table 8.1: Typical layout of an SLC block

Row Index	First 2^{14} cells	Last 2^{14} cells
0	page 0	page 1
1	page 2	page 3
2	page 4	page 5
⋮	⋮	⋮
30	page 60	page 61
31	page 62	page 63

Table 8.2: Typical layout of an MLC block

Row Index	MSB of the first 2^{15} cells	LSB of the first 2^{15} cells	MSB of the last 2^{15} cells	LSB of the last 2^{15} cells
0	page 0	page 4	page 1	page 5
1	page 2	page 8	page 3	page 9
2	page 6	page 12	page 7	page 13
3	page 10	page 16	page 11	page 17
⋮	⋮	⋮	⋮	⋮
30	page 118	page 124	page 119	page 125
31	page 122	page 126	page 123	page 127

In order to reduce the number of block erasure operations, an updated version of a stored page is simply written into another available physical location, and its previous location is marked as invalid. A table, called the Flash Transition Layer (FTL) [1], keeps a record of the latest mapping between logical and physical pages and is maintained in the memory device. When the memory becomes full (or reaches a pre-specified storage capacity), blocks no longer in active use need to be erased to allow new data to be stored. To enhance device lifetime, "wear-leveling" algorithms are used to balance the number of erasures among blocks within a single device [1].

Each page in a flash memory block contains a spare area. If the page size is 2KB then a typical spare area can be 64B. A portion of this spare area is used to store metadata in order to build the FTL once the flash memory is activated. The rest of the spare area is dedicated to storing the redundancy bytes of the error-correcting codes [2].

Remark 8.2.1. The organization of pages in a flash memory block may differ from one manufacturer to another. The configurations shown in Tables 8.1 and 8.2 are consistent with the information available to us about the devices tested, as well as with most of the results of our experiments.

8.3 Error Characterization of Flash Memories

In order to have a basic characterization of the error behavior in flash memories we repeated the process of erasing, writing pseudo-random data, and reading to compare and find errors for SLC and MLC blocks. The raw BER as a function of the program/erase cycle of the blocks is given in Fig. 8.2 and Fig. 8.3. We now use the results of these experiments to gain further understanding about the error characteristics and mechanisms in these chips.

8.3.1 Page-level BER

We also examined the BER of each individual page in the block in order to determine if the BER has any significant page dependency. The page-level BER measurements were used to generate a three-dimensional picture, as shown in Fig. 8.4 for the SLC block and in Fig. 8.5 for the MLC block. The raw BER as a function of the program/erase cycle is given for each page individually.

In the SLC case, we observed that in general the BERs of the pages in the left-hand part of the block are significantly larger than those of the pages in the right-hand part. One possible

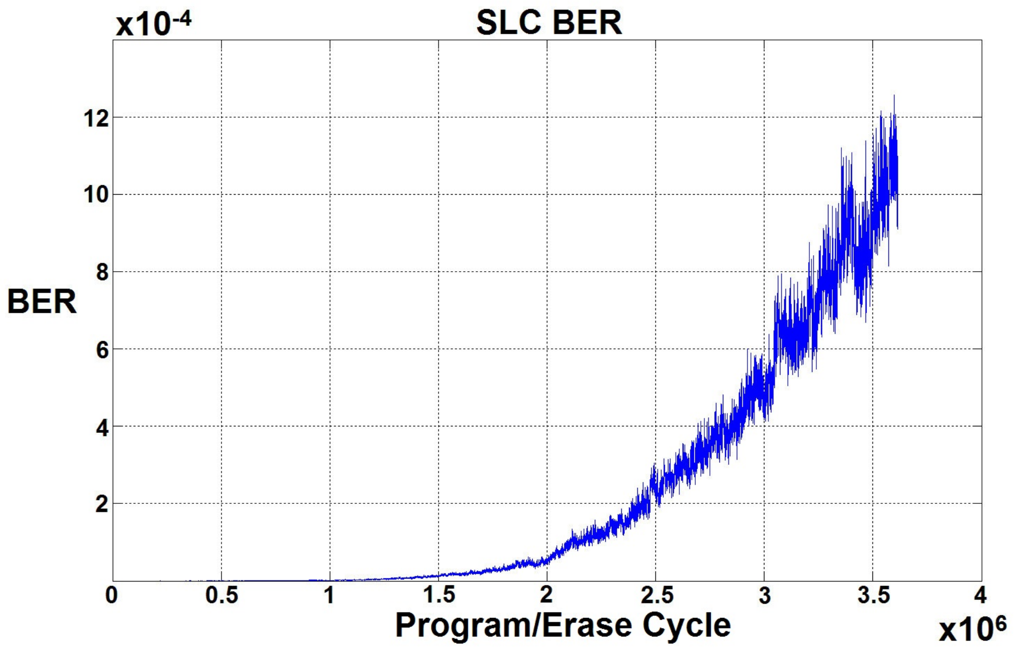


Figure 8.2: Raw BER for SLC flash.

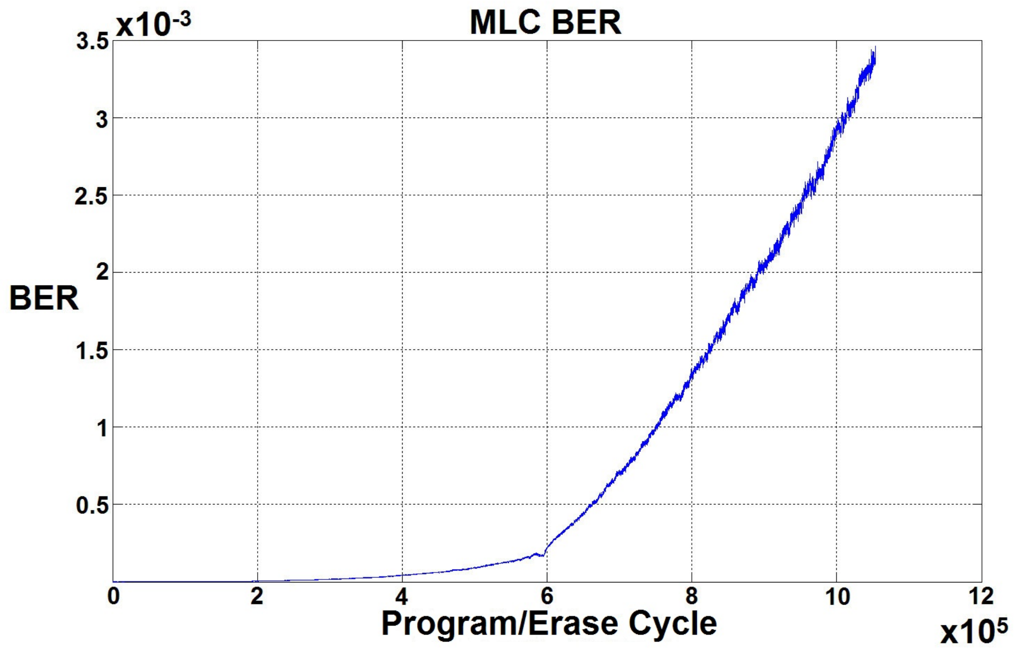


Figure 8.3: Raw BER for MLC flash.

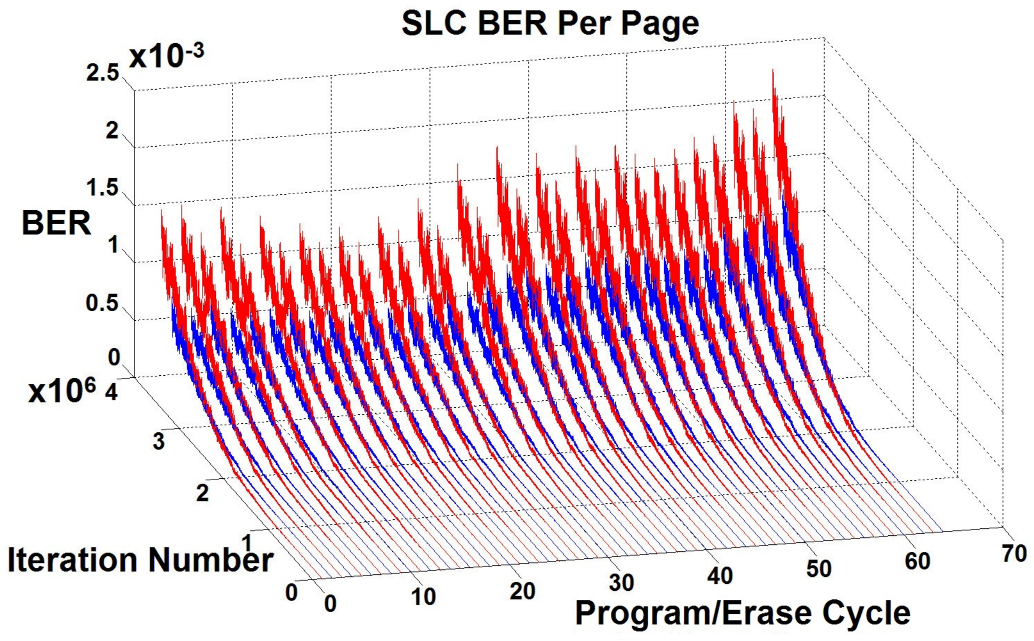


Figure 8.4: BER per Page for SLC Block.

explanation for this phenomenon is related to the way in which the cells are programmed. In each row, the left-hand page is programmed first, followed by the programming of the corresponding right-hand page. We speculate that the programming of the right-hand page somehow disturbs the cells in the left-hand page, inducing more errors when the left-hand page is later read.

In the MLC case, we see that the LSB pages generally have a higher BER than the MSB pages. This is due to the fact that the assignment of 2-bit patterns to threshold values within a cell makes the LSB more susceptible to errors than the MSB. This will be discussed in more detail in Section 8.5. We also noted anomalous BER characteristics in the first and last few pages of the MLC block. We have not yet found an explanation for this behavior.

8.3.2 Bit-level BER

Next, we investigated the error performance at the bit level within each block. The number of errors in each bit was accumulated over a small number of consecutive program/erase cycles during which the error statistics could be assumed to be fairly constant. For the SLC block, we considered cycles 1.5×10^6 to 1.6×10^6. The total number of errors in each bit location was counted and the results were plotted in a three-dimensional histogram, shown in Fig. 8.6. In the

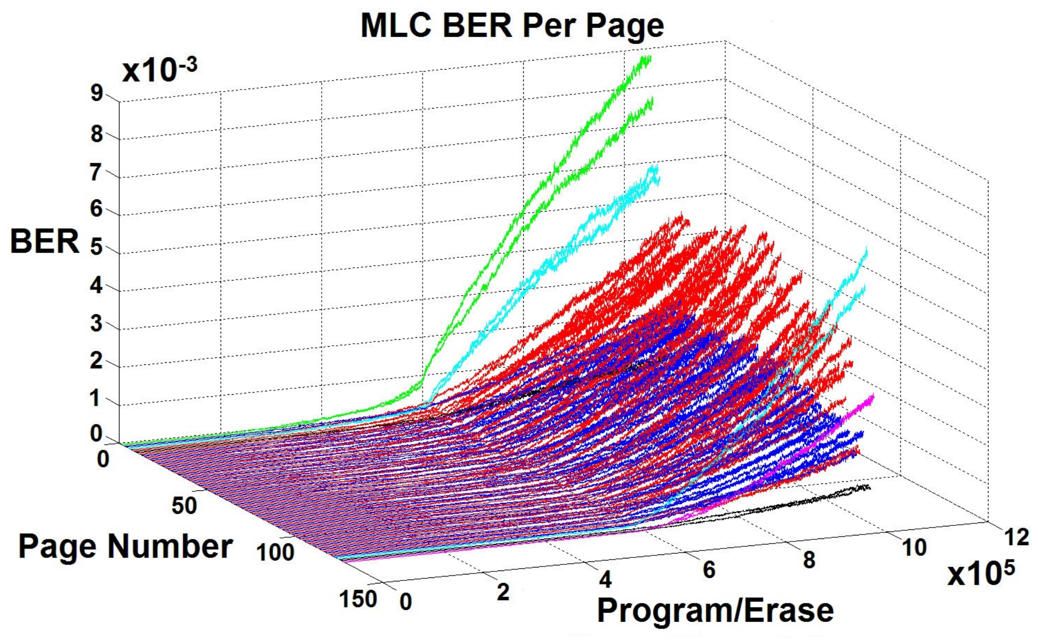


Figure 8.5: BER per Page for MLC Block.

figure, the results for the left-hand pages and right-hand pages are shown separately. It can be seen that the errors are clustered in columns rather than rows. We can also see that there are fewer errors overall in the right-hand part of the block, as is to be expected from Fig. 8.4. Bit-level BER measurements for the MLC blocks (not shown here) displayed different characteristics. In contrast to the SLC case, we did not find that the errors were clustered along the bit lines; rather, the bit-error locations appear to be randomly distributed among the cells.

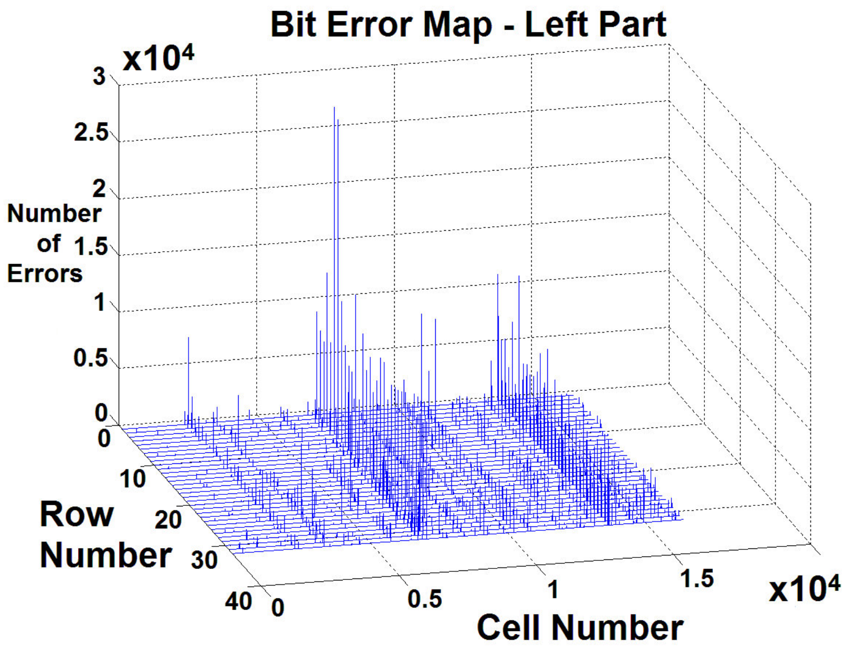
8.4 Partial Cell State Usage in MLC

Even though MLC flash memories can increase storage capacity, they tend to be less reliable and to have shorter lifetimes. In this section, we wish to explore the use of an MLC device as if it were SLC. The idea is that this will provide additional immunity to errors in the stored information.

There are several ways in which one could store only a single bit in MLC flash:

1. Program only the MSB pages.

2. Program only the LSB pages.



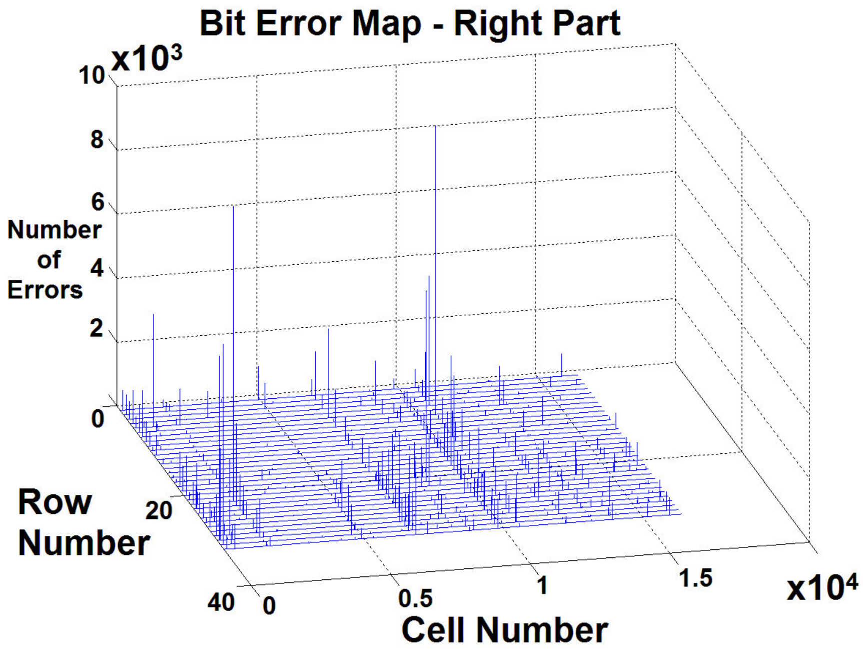


Figure 8.6: Bit Error Map for SLC Block.

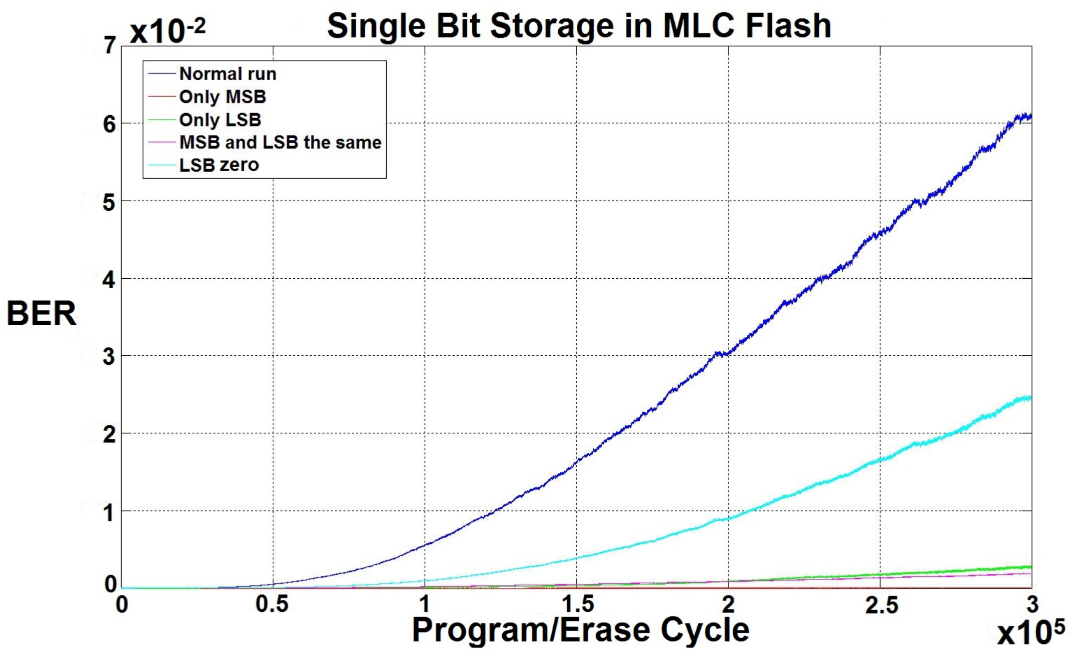


Figure 8.7: Different Schemes for Storing a Single Bit in an MLC Block.

3. Program the LSB and MSB pages with the same values. (The cells will therefore be in state 11 or 00.)

4. Program the data in the MSB pages, and program all LSB pages to all-0 bit values. (The cells will therefore be in state 11 or 01.)

Fig. 8.7 shows the measured BER for all four of these schemes. The results show that the best approach is the first one, in which we program only the MSB pages.

This technique can also be made adaptive by initially using the MLC device in its native mode, and then switching to SLC mode after a certain number of iterations, when the BER has become larger due to device wear. Experimental results confirmed that this mode of operation can indeed enhance the MLC device endurance.

8.5 ECC for MLC Flash

In this section, we give a more complete characterization of the errors in MLC flash blocks. We then propose a new ECC scheme designed to correct the dominant errors. For MLC flash blocks, we want to determine the most likely transitions between the four states that a cell can support.

To that end, we collected the errors found as we iterated the operation of erasing a block, programming pseudo-random data, and then reading back the data. Using the MLC block layout shown in Table 8.2, we then characterized the observed error types in terms of cell state transitions. In theory, an error corresponding to any change in cell level is possible. However, we found in our experiments that the different error transitions are not equally likely. Rather, the dominant errors were those in which the cell voltage changed by one level, particularly from state 10 to state 00 or from state 00 to state 01. Errors where the voltage changed by two or three levels were very rare.

These results suggest a new ECC scheme for MLC flash. Today, the ECC in MLC flash operates on individual pages. That is, even though an MSB page and an LSB page share the same group of cells, the errors in each page are corrected independently. The idea behind the new ECC scheme is to focus on correcting the dominant single-level cell-state errors by sharing some redundancy between the pair of MSB and LSB pages.

Let $\mathcal{C}_1$ be a t_1-error-correcting BCH code and let $\mathcal{C}_2$ be a t_2-error-correcting BCH code, where $t_2 > t_1$. We use systematic encoders for both codes, and we choose them to be "compatible" in the following sense. Suppose that for a given information word, the encoder for $\mathcal{C}_1$ generates r_1 redundancy bits and the encoder for $\mathcal{C}_2$ generates r_2 redundancy bits. Then, we assume that by the r_2 redundancy bits generated by the encoder for $\mathcal{C}_2$ it is possible to calculate the r_1 redundancy bits generated by the encoder for $\mathcal{C}_1$.

Let $\boldsymbol{p}_{\text{MSB}} = (a_0, \ldots, a_{n-1}), \boldsymbol{p}_{\text{LSB}} = (b_0, \ldots, b_{n-1})$ be an MSB page and an LSB page sharing the same group of cells. The encoding proceeds as follows.

Encoding:

1. Calculate $\boldsymbol{s}_1$, the r_1 redundancy bits of $\mathcal{C}_1$ corresponding to the information page $\boldsymbol{p}_{\text{MSB}}$.

2. Calculate $\boldsymbol{s}_2$, the r_2 redundancy bits of $\mathcal{C}_2$ corresponding to the information page $\boldsymbol{p}_{\text{MSB}} + \boldsymbol{p}_{\text{LSB}}$.

Note that, by the linearity of the codes $\mathcal{C}_1$ and $\mathcal{C}_2$, the r_1 redundancy bits of $\mathcal{C}_1$ corresponding to the information page $\boldsymbol{p}_{\text{LSB}}$ are the sum of $\boldsymbol{s}_1$ and the r_1 redundancy bits that can be computed by $\boldsymbol{s}_2$. Therefore, it is possible to correct t_1 errors in the LSB page, as well. The decoding procedure, which we now describe, makes use of this property.

Decoding:

1. Using the r_2 bits corresponding to $\boldsymbol{s}_2$ and a decoder for the code $\mathcal{C}_2$, find up to t_2 errors in

$p_{\text{MSB}} + p_{\text{LSB}}$.

2. Change the states of the cells identified as erroneous by the $\mathcal{C}_2$ decoder by one level, according to the rule: state 11 is changed to state 10, and vice versa, state 00 is changed to state 10, and state 01 is changed to state 00.

3. Using the r_1 bits corresponding to s_1 and a decoder for the code $\mathcal{C}_1$, find up to t_1 errors in the page p_{MSB}.

4. Compute the sum of the r_1 bits corresponding to s_1 and the associated r_1 bits that are computed from s_2. Using this sum and a decoder for the code $\mathcal{C}_1$, find up to t_1 errors in p_{LSB}.

We applied this ECC scheme to the MLC flash device and compared its page-level performance to that of a BCH code. Fig. 8.8 shows the results for a BCH code that corrects 20 errors in each page and the new ECC scheme, where the code $\mathcal{C}_2$ can correct 35 errors and the code $\mathcal{C}_1$ can correct 5 errors. With these parameters, the two coding schemes have the same overall redundancy. In the error-rate evaluation, we assume for each code that decoding is successful if the number of errors in a codeword is no greater than the code's specified error correction capability. If the number exceeds the correction capability, we assume this condition is detected, and the received word remains unchanged. Our results show that for sufficiently low or sufficiently high raw error rates, the two coded behave similarly. However, when the number of errors is in the range typically found after $32,000$ to $40,000$ program/erase cycles, the new coding scheme can correctly decode both the MSB and the LSB pages, while the BCH code tends to fail, as shown in Fig. 8.8.

Remark 8.5.1. Our goal in this section was to suggest the possibility of a new ECC scheme for MLC flash memories that simultaneously protects MSB and LSB pages. The performance evaluation is based upon certain assumptions about the decoder that need to be more carefully assessed. In particular, a full analysis should consider the vulnerability of the scheme to mis-correction by the $\mathcal{C}_2$ decoder. We also note that the code design was motivated by the error data collected from a particular MLC chip in a controlled laboratory environment. A more thorough evaluation of the scheme would have to take into account device variability arising from different manufacturers (see, for example, [4]) and operating conditions.

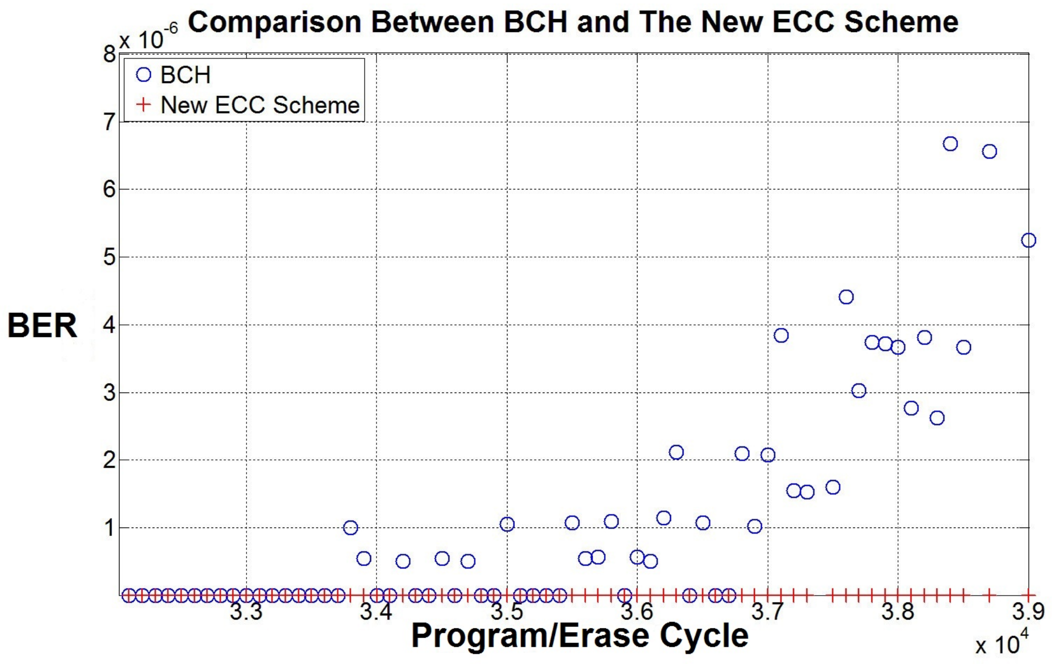


Figure 8.8: Comparison Between ECC schemes.

8.6 WOM-Codes for Flash Memories

In this section, we discuss the implementation of WOM-codes which were extensively studies in Section 3 and 4. These codes are intended to permit reuse of memories whose cells can be programmed only once, such as punch cards or optical storage memories. In [5], a simple WOM-code that can store two bits twice using three cells was presented. Table 3.1 gives the encoding and decoding rules for this WOM-code.

A block in a flash memory chip is very similar to a write-once memory in that a cell level can only be increased, not decreased, on successive writes, unless the entire block is erased. This observation suggests the use of WOM-codes in flash memories as a means of reducing the number of block erasures and, accordingly, increasing the device endurance.

As an example, we describe the application of the two-write WOM-code in Table 3.1 to an SLC flash memory. For every 2KB page in a block, we encode $\lfloor 2\text{KB}/1.5 \rfloor = \lfloor 4/3\text{KB} \rfloor$ of data using the rate-2/3 WOM encoder. The encoded pages are written successively into the block using a conventional wear-leveling algorithm, but with a slight modification. Specifically, when the wear-leveling algorithm calls for a block to be erased, the block is not physically erased but, rather, is marked as “invalid.” In subsequent page-write operations to the block, the wear-

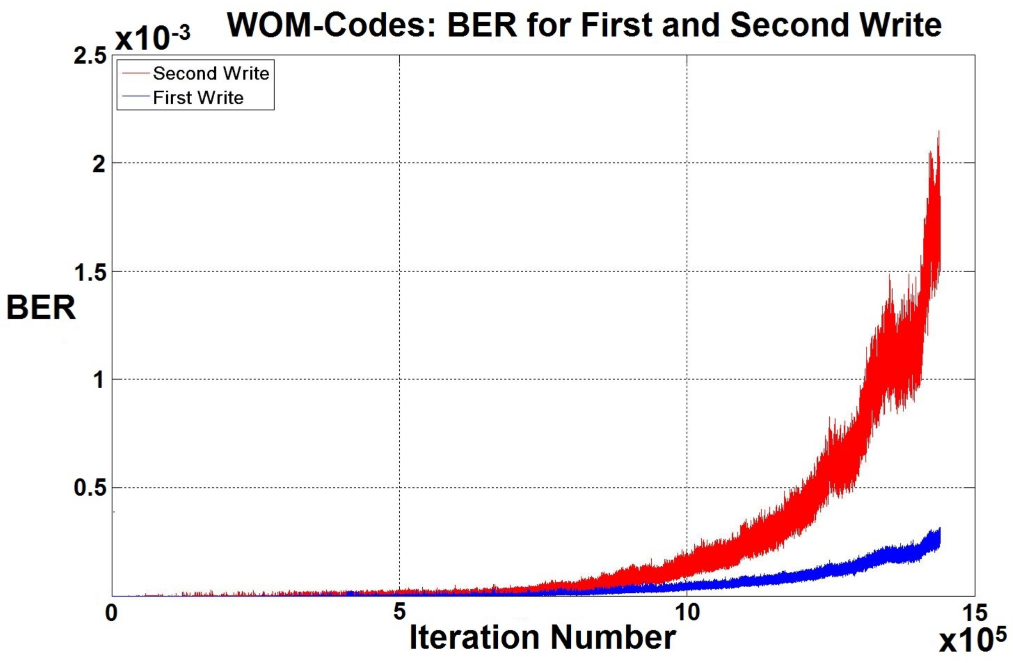


Figure 8.9: BER for first and second write of WOM-codes in SLC flash.

leveling algorithm treats the block as if it were erased, but first reads the page to be rewritten, and then makes use of the encoding rules for the second write of the WOM-code to program the page. Such rewrite operations can continue until the wear-leveling algorithm determines that the block must be erased, at which point the block is physically erased, and the programming cycle begins anew.

We implemented this WOM-code programming scheme on an SLC device and investigated the effect of the second write on the block BER. The results are shown in Fig. 8.9. The block BERs associated with the first and second writes are shown in blue and red, respectively, as a function of the iteration number. Clearly, the BER of the second write exceeds that of the first write, and quite substantially after about 8×10^5 iterations. However, below 5×10^5 iterations, the increase in BER appears to be small, and it appears that the degradation in performance could be offset by the use of an appropriate error correction strategy with minimal extra overhead. Further discussion of WOM-codes and their application to MLC flash memory can be found in [3].

8.7 Summary and Conclusions

In this work, we used empirical data to investigate the characteristics of errors in SLC and MLC flash memory devices. We studied the error behavior at the block level, page level, and bit level. Our observations motivated the design of a new error-correcting scheme for MLC flash that outperforms conventional BCH codes. We also measured the improvement in BER that could be achieved with a reduction in the nominal chip storage capacity. Finally, we described the application of WOM-codes to flash memories, and we experimentally evaluated the performance of a simple WOM-code on an SLC device.

Acknowledgment

This chapter is in part a reprint of the material in the paper: E. Yaakobi, J. Ma, L. Grupp, P.H. Siegel, S.Swanson, and J.K. Wolf, "Error Characterization and Coding Schemes for Flash Memories," *Proc. Workshop on the Application of Communication Theory to Emerging Memory Technologies*, Miami, Florida, December 2010.

Bibliography

[1] E. Gal and S. Toledo, "Algorithms and data structures for flash memories," *ACM Computing Surveys*, vol. 37, pp. 138–163, June 2005.

[2] S. Gregori, A. Cabrini, O. Khouri, and G. Torelli, "On-chip error correcting techniques for new-generation flash memories," *Proceedings of The IEEE*, vol. 91, no. 4, pp. 602–616, April 2003.

[3] L. Grupp, A. Caulfield, J. Coburn, S. Swanson, E. Yaakobi, P.H. Siegel, and J.K. Wolf, "Characterizing flash memory : anomalies, observations, and applications," *MICRO-42*, pp. 24–33, December 2009.

[4] N. Mielke, T. Marquart, N. Wu, J. Kessenich, H. Belgal, E. Schares, F. Triverdi, "Bit error rate in NAND flash memories,"in *Proceedings of IEEE Reliability Physics Symposium*, pp. 9–19, May 2008.

[5] R.L. Rivest and A. Shamir, "How to reuse a write-once memory," *Information and Control*, vol. 55, no. 1–3, pp. 1–19, December 1982.