WORST-CASE AND AVERAGE-CASE FLOATING CODES FOR FLASH MEMORY

by
Hilary Finucane

under the guidance of
Professor Michael Mitzenmacher
Harvard University

A thesis
presented to the Department of Mathematics
in partial fulfillment
of the requirements for the degree
of Bachelor of Arts with Honors

Harvard University
Cambridge, Massachusetts
March 20, 2009

Abstract

Flash memory is used in portable electronic devices like cell phones, mp3 players, digital cameras, and PDAs. The question of how to code information in flash memory is a difficult one whose answer can have effects on the speed and lifespan of a chip of flash memory. The main factor to be taken into account when designing codes for flash memory is the difference in cost between decreasing and increasing the state of a cell; decreasing the state of a cell requires first the resetting of an entire section of cells. Codes that focus on maximizing the number of updates between these reset operations are called floating codes. Worst-case floating codes focus on maximizing the lower bound on the number of updates between resets, and average-case floating codes focus on maximizing the average number of updates between resets. In this thesis, we review the work done on both worst-case and average-case floating codes. We also introduce a new code which performs as well as the previous best worst-case floating code but is significantly simpler, and a second code which performs asymptotically better.

Acknowledgments

Thanks to Professor Mitzenmacher for his guidance and advice. From helping me pick a topic to the final edits, he has always been helpful, supportive, and generous with his time. Thanks also to Thomas Finucane, Mariel Finucane, Avner May, Rishi Gupta, Yakir Reshef, Mark Browning, and Shira Mitchell for their helpful comments.

Contents

Chapter 1

Introduction

1.1 Motivation

Nonvolatile memory is computer memory that maintains stored information without a power supply. For example, the now ancient punch card is a type of nonvolatile memory because, though it requires power to punch, it does not require power to remain punched. With the rise of portable electronic devices like cell phones, mp3 players, digital cameras, and PDAs, nonvolatile memory is increasingly important. Flash memory is currently the dominant nonvolatile memory because it is cheap and, unlike punch cards and other more recent kinds of nonvolatile memory, can be electrically programmed and erased with relative ease [2]. In the fourth quarter of 2008, flash memory revenue totaled over two billion dollars [6].

A chip of flash memory contains an array of tens of thousands of cells, and we assume that each chip stores a bit string. Each cell can be in one of multiple states, and the states of the cells encode the bit string stored on the chip; if the bit string stored on the chip changes, then the states of the cells must also change to encode the new bit string correctly. But which cell state configurations should correspond to which bit strings, and how the cell states should change as the bit string changes are complicated and subtle questions. They are also important questions, since efficient encoding methods can increase the speed and lifespan of flash memory [13]. Efficient encoding techniques for flash memory are the topic of this thesis.

1.2 Floating Codes: An Introduction

Each cell on a chip of flash memory can be thought of as a container for electrons. In *binary flash*, each cell has two states: if there are electrons in the container then the cell is in state 1, and if there are no electrons in the container, the cell is in state 0. Until recently, binary flash was the only kind of flash available, but now a new kind of flash memory has been developed, *multilevel flash*, that many see as the future of flash memory. In a multilevel cell, it is possible to distinguish between several different ranges of charge, allowing for more than two states [3].

To transition between states, it is necessary to add and remove electrons to and from the container. While it is easy to add electrons (i.e. to increase the state of the cell), it is impossible to remove electrons (i.e. to decrease the state of the cell) without first emptying the electrons from all of the containers in a large section of the chip; this process, called a *reset operation*, is slow and, after many repetitions, wears out the chip [3]. In multilevel flash, there are two types of mistakes that can occur when programming a cell: errors in which too many electrons are added ("overshoots"), and errors in which too few electrons are added ("undershoots"). Because of the difficulty of removing electrons, overshoots are a much bigger problem than undershoots, which are easily corrected by adding electrons. To avoid overshoots, the level of a cell is increased over multiple iterations by carefully adding small numbers of electrons at a time [15]. For more details on how flash memory works, see Appendix 1.

When designing an encoding scheme for multilevel flash memory, there are many factors to take into account, such as the possibility of error and the multiple iterations needed for programming. However, the most salient factor by far is a factor shared with binary flash: the difference in cost (in terms of efficiency and the lifespan of the chip) between increasing and decreasing the state of a cell. Since it is much more costly to decrease than to increase the state of a cell, decreasing cell states should be avoided as much as possible. Thus, one property of a good encoding scheme is that the states of cells are seldom decreased — that is, that reset operations are rare. Encoding schemes that try to maximize the number of updates between reset operations, assuming no errors will occur, are called *floating codes* [13].

For example, consider the case where one bit is encoded in one cell, and the cell has six states numbered 0 through 5. The bit and cell both begin in state 0. The goal is to find a way to assign each cell state to a 0 or a 1 in such a way that the value of the bit can be updated many times ("flipped" from 0 to 1 or 1 to 0), with the state of the cell never decreasing but always decoding to the correct value of the bit, before the cell has to be reset. If you assign cell states 0, 1, and 2 to bit

Bit Strings	Cell States After One Update	Cell States After Two Updates
00		111
01	100	011
10	010	101
11	001	110

Figure 1.1: A code for two bits and three-two level cells, guaranteeing two updates from 00 without resetting [16].

value 0, and states 3, 4, and 5 to 1, then when the bit is flipped once, the cell will switch from state 0 to state 3 — the nearest state that decodes to a 1 — and when it is flipped again the cell will have to reset, since no states higher than 3 will decode as 0. So this encoding scheme guarantees only one update before the cell is reset. Suppose instead that you assign all the even states to bit value 0, and all the odd states to bit value 1. Then every time the bit flips, the cell state increases by only 1 to reflect the new value of the bit. In this encoding scheme, we have guaranteed 5 updates before a reset. The second of these two codes is considered a better floating code than the first, because it guarantees more updates.

Note, however, that if the goal were not to maximize the number of updates before a reset, but to have a code that was able to maintain accuracy even if a cell was accidentally programmed to a state higher than intended, then the first code would be preferable because a small overshoot, such as setting the cell to state 4 or 5 instead of state 3, would not cause the cell to record the wrong value. Although the second code is a better floating code, it will not record the right value if there is an overshoot.

One example of a floating code for the case of two bits and three two-state cells (again assuming that all bits and cells start as zeros) is due to Rivest and Shamir [16]. Their code is presented in Figure 1.1. A two-bit string begins as 00, and can be updated to any other two-bit string. After this first update, the new string is encoded into the corresponding cell states from the second column; when the bit string is updated a second time, the new string can be encoded into the corresponding cell states from the third column, without decreasing the state of any cell. For example, if the bit string starts as 00, then is updated to 11 and then 01, the cell states will change from 000 to 001 to 011. We say that this code guarantees two updates without resetting.

In general, the number of cells in a section is not one or three, but closer to 2^{17} or 2^{20} [19], and the number of states per cell can range from 2 to 256, so these examples are much simpler than the more general codes we will be presenting in later chapters. However, we will continue to assume

that the information being stored is a string of bits of fixed length.

1.3 Floating Codes: A Formal Definition

Formally, we define a floating code as follows. There are n cells in states $c_0, \ldots, c_{n-1}$, $c_i \in \{0, \ldots, q-1\}$. We call $(c_0, \ldots, c_{n-1})$ the *cell state vector*. We store k variables with values $v_0, \ldots, v_{k-1}$ in these cells, where $v_i \in \{0, 1\}$. We call $(v_0, \ldots, v_{k-1})$ the *variable vector*, or the bit string. We often refer to v_b as the value of bit b. (In other formulations of the problem, $v_i \in \{0, \ldots, l-1\}$ for some $l \geq 2$, but we focus only on the case where our variables are bits.) The *weight* of a set of cells is the sum of their states.

A worst-case floating code is made up of two mappings: an update function and a decode function. According to the most common definition of a floating code, only one bit flips at a time, so the update function $U : \{0, \ldots, q-1\}^n \times \{0, \ldots, (k-1)\} \to \{0, \ldots, q-1\}^n \cup \{\perp\}$ takes the current value of the cell state vector, together with the bit that is flipped, and returns the updated value of the cell state vector, or returns $\perp$ if a reset operation is called for. The update function cannot decrease the state of any cell. The decode function $D : \{0, \ldots, q-1\}^n \to \{0, 1\}^k$ returns the values of the variables encoded in a given cell state vector. These two functions must satisfy the property that if $U(s, b) = t$, then $D(s) \oplus D(t) = e_b$, where e_b is the bit string of all zeros except for a one in the b^{th} position, and $\oplus$ denotes the XOR operator, or element-wise addition mod two. In other words, the bit strings $D(s)$ and $D(t)$ differ only in the b^{th} position. In an alternate formulation of the problem, more than one bit can flip at a time; see, for example, the code for two bits and three two-state cells from Section 1.2. This second type of code is discussed in more detail in Section 4.5. Worst-case floating codes focus on maximizing the lower bound on the number of updates between reset operations. Only three papers have been published focusing on worst-case floating codes as defined here ([13], [14], [19]); we discuss these papers in Chapter 2.

An average-case floating code is also made up of an update function and a decode function, but the update function allows the state of a cell to decrease, imposing a unit of cost whenever this happens; $\perp$ is never used. The cost per bit flip is averaged over time, based on an underlying probabilistic model, and the goal is to minimize this average cost. This formulation was introduced recently [5], and is presented in more detail in Chapter 4.

1.4 Other Codes for Flash Memory

In this thesis, we will focus on floating codes, trying to maximize the number of updates between reset operations, assuming no errors. Even with the simplifying assumption that no errors will occur, the design of floating codes is an interesting theoretical problem, and there is also a lot to be learned from the study of floating codes that is important in the design of real-world codes for flash memory. It should be noted, however, that there are a variety of other codes that address other problems specific to flash memory, such as identifying and fixing the asymmetrical errors induced when a cell is accidentally programmed to a state higher than intended [4]. For example, if a code only uses the even states of a cell, it is possible to detect any small overshoots by noting that the cell is in an odd state. These errors can be corrected by increasing the cell state to the closest available state that encodes the same bit value as is encoded by the state immediately below the current odd state. Other error-correcting codes focus on errors in reading and writing, errors caused by slow loss of charge over time, or errors caused by accidental charging of neighboring cells. Several papers ([4], [9], [12]) have been written on error-correcting codes for flash memory.

Another class of codes, rank modulation codes, has challenged the idea that the number of states should be fixed, arguing that codes should be based on relative, analog cell states, rather than absolute states. That is, instead of fixing several ranges of electrons in the container, these codes are only based on which containers have more electrons than which other containers. These codes guarantee fewer updates than floating codes, but they lessen the need for precise addition of electrons into the container, allowing for much faster write operations and fewer errors [15]. While this thesis focuses mostly on floating codes, a goal for future work is to combine these various approaches into a code that guarantees many updates, while still allowing for error correction and speed.

1.5 Outline

In Chapter 2, I will review some predecessors of floating codes and discuss the previous work done on the topic of worst-case floating codes. In Chapter 3, I will present two original codes, designed for the worst case. The first performs asymptotically as well as the current state of the art in floating codes, but is significantly simpler and has the potential to be more easily adjusted for error correction and other models. The second code performs asymptotically better than the best previously known floating code. In Chapter 4, I will focus on average-case floating codes, describing the formal model and some principles of average-case construction, with examples. In Chapter 4, I will also compare worst-case and average-case floating codes under the subset-flip model, where more than

one bit can flip at a time. In Chapter 5, I will outline directions for future research.

The work presented in Chapter 3 is my original work, and is the main contribution of this thesis. Chapters 2 and 4 are original exposition of work that has been previously published. The results of Chapter 4 were developed in collaboration with Flavio Chierichetti, Zhenming Liu, and Michael Mitzenmacher [5].

Chapter 2

Worst-Case Floating Codes

The field of floating codes is quite new; the term "floating code" was coined in 2007 [13]. However, floating codes have roots in the older field of Write-Once Memory (WOM) codes, which correspond to floating codes where each cell has two states. In Section 2.1, we discuss WOM codes and the first two papers published on floating codes. In Section 2.2, we discuss in detail a third paper, which raised the standard for floating codes and provided a basis for the code we present in Chapter 3.

2.1 Predecessors and Early Codes

WOM Codes

In 1982, Ronald Rivest and Adi Shamir published a seminal article, "How to reuse a 'write-once' memory" [16], founding the field of WOM codes and, more generally, introducing the idea that write-asymmetric memory could be used non-trivially. They defined *wits* as write-once cells with two states (for example, punch cards), and opened with the lemma that only three wits are needed to "write two bits twice;" i.e. to record, without resetting any cells, a two-bit value that changes to another two-bit value. (See Section 1.2 for a presentation of this code.)

Following this groundbreaking presentation of the first floating code, Rivest and Shamir introduce several new concepts and prove several new results. In particular, they prove that only $t + o(t)$ wits are needed to write k bits t times for any fixed k; that only $\frac{kt}{\log t}$ wits are needed for any fixed t as k gets large; and that for an array of wits of fixed size n in which k bits can be written t times, kt can get as large as $n \log n$. However, because these results are not directly relevant to floating codes as they are currently studied, we will not describe them in detail.

A number of other papers build on the ideas of Rivest and Shamir ([1] [7] [8] [10] [11] [17] [18]). Some propose new codes for the same model, while others generalize the WOM code to include, for example, larger alphabets, or different restrictions on state transitions.

At the time these papers were published, multilevel flash had not yet been invented, and so the idea of multiple states per cell was only mentioned as one of several possible generalizations of WOM codes. With the invention of multilevel flash several years later, however, this particular generalization of WOM codes suddenly rose in importance, and the focus shifted from WOM codes to floating codes.

Jiang, et al.

In 2007, Anxiao Jiang, Vasken Bohossian, and Jehoshua Bruck defined floating codes. In their two papers on floating codes ([13], [14]), they introduce the specific model described in the introduction, including the new variable q (number of states per cell). They also reformulate the problem, optimizing t for a fixed n instead of n for a fixed t. This formulation is more relevant for the designers of flash memory, who have a fixed number of cells they are hoping to use as efficiently as possible. Over the course of these two papers, they introduce several codes for limited values of k and n and a method for extending these codes to larger values of k and n.

For example, they introduce a code for $k = 2$, $n \geq 3$, and arbitrary q, as follows. The notation a^b refers to b cells in a row, all of which are in state a. At all times during the encoding process, there exist non-negative integers i, x, y, and z, with $y > 0$ such that the cell state vector is of the form $(i+1)^x i^y (i+1)^z$. Decoding, we get that $v_0 = x \pmod 2$, and $v_1 = z \pmod 2$. If $y > 1$, then if bit 0 flips, the cell state vector is updated to $(i+1)^{x+1} i^{y-1} (i+1)^z$ and if bit 1 flips, the cell state vector is updated to $(i+1)^x i^{y-1} (i+1)^{z+1}$. If $y = 1$ then if either bit flips, the cell state vector is updated to be $(i+2)^{v_0} (i+1)^{n-v_0-v_1} (i+2)^{v_1}$. For example ($n = 5$), if the bit string starts as 00, then is updated as follows: 00, 01, 11, 10, 11, 10, 00, 01 the cell states would change as follows: 00000, 00001, 10001, 10011, 10111, 21111, 22111, 22112.

For the above construction, we say the cell state vector is in *phase* a if all cells are in state a or $a - 1$. Jiang et al. show that $n - 1$ updates occur in phase one, and at least $n - 2$ in each other phase (where an update that changes the cell state vector from phase a to phase $a + 1$ counts as an update occurring in phase $a + 1$). There are $q - 1$ phases between reset operations, so this code guarantees $t = (n - 2)(q - 1) + 1$ updates.

Jiange et al. present several other codes for limited values of k and n (ex. $k = n$, or $3 \leq k \leq 6$). Their most significant construction, though, is an *indexed code*, which allows them to extend codes for limited values of k and n to codes for arbitrary k and n. The idea behind the indexed code is to break the cells and the bits into groups and store different groups of bits in different groups of cells, keeping track of which groups of bits were recorded in which groups of cells through an indexing scheme. They show that this construction also guarantees $t = n(q-1) - o(nq)$ updates, but for any values of k, q, and n.

To measure the performance of their codes, they first note that the trivial upper bound on the guaranteed number of updates of any floating code is $n(q-1)$, because each time a bit is flipped at least one cell must increase by at least one state, and this can only happen $n(q-1)$ times before all cells are in state $q-1$. Since $\lim_{n\to\infty} \frac{t}{n(q-1)} = 1$ for their codes, and no code can guarantee more than $n(q-1)$ writes, their codes are *asymptotically optimal*.

They also prove two upper bounds on t. The first is as follows: if $n \geq k-1$, then $t \leq (n-k+1)(q-1) + \lfloor \frac{(k-1)(q-1)}{2} \rfloor$; if $n < k-1$, then $t \leq \lfloor \frac{n(q-1)}{2} \rfloor$. The second is quite complicated and not relevant to this thesis.

2.2 Yaakobi et al.

In 2008, a paper by Eitan Yaakobi, Alexander Vardy, Paul Siegel, and Jack Wolf was published [19], with a new code that improved considerably on the codes by Jiang et al. In this section, we will discuss the two main contributions of this paper: a new measure for how good a code is called *deficiency*, and their new floating code.

Unused Levels and Deficiency

If a set S of m cells has weight w, then we refer to $m(q-1) - w$ as the number of *unused levels* of S. This refers to the fact that states of cells of S could be incremented $m(q-1) - w$ more times before all cells of S were at state $q-1$. Combining the idea of unused levels with the trivial upper bound $n(q-1)$ on the guaranteed number of updates of a floating code, Yaakobi et al. define the deficiency of a code to be $n(q-1) - t$. (Note that this new measure allows for codes that are asymptotically better than the "asymptotically optimal" codes of Jiang et al.) The indexed code of Jiang et al. has deficiency dependent on n, but the paper by Yaakobi et al. introduces a code for arbitrary n, q, and k, called the Enhanced Multidimensional Construction (EMC), with deficiency $O(k^2 q)$. According

to the authors, n is typically around 2^{17} or 2^{20}, whereas in all practical situations, $q \leq 256$, so this code represents a significant improvement over previous codes. In Section 2.2.2, we present a slightly simplified version of their code. A more precise description of the code as presented in the paper follows in Section 2.2.3.

EMC: A Simplified Version

The code described in this section has the same asymptotic deficiency as the code by Yaakobi et al., but is slightly simpler and less efficient; we describe it to make the efficient version easier to understand. The code is constructed recursively on k, and assumes odd q.

The Base Case

We start by describing a code for $k = 2$ and arbitrary n and q. We divide the cells into $\frac{n}{2}$ *blocks* of two cells each, which we picture as being arranged from left to right. We call a block *empty* if it has weight 0, *full* if each cell in the block is in state $q-1$, and *active* otherwise. We also refer to cells as being empty, full, or active. Each block will eventually be assigned to record the successive values of either bit 0 or bit 1, and there is at most one active block assigned to each bit at a time. If bit 0 (respectively, bit 1) flips, then we find the active block assigned to bit 0 (resp. bit 1), if there is one, and increase the state of the left-most (resp. right-most) non-full cell in that block by one. If no active block is assigned to the bit, then we assign the left-most empty block to the bit and then increase the state of the left (resp. right) cell by one. So in an active block assigned to bit 0 (resp. bit 1), the left (resp. right) cell is always in a state greater than the state of the right (resp. left) cell, so we can always tell which active block is assigned to which bit.

To decode, we find the active block(s) and decode which bit each active block is assigned to by comparing the states of the two cells in the block. If no active block is assigned to bit b for some b, then $v_b = 0$. If the active cell in an active block assigned to bit b is in state x, then $v_b = x \pmod 2$. If the active block assigned to bit b has no active cell, then $v_b = 0$. Note that the ability to ignore all full cells when decoding relies on the fact that q is odd and that it takes $q-1$ bit flips to fill a cell. See Figure 2.1 for an example where $n = 10$ and $q = 5$.

The Recursion

We now have a code for $k = 2$ and arbitrary n and q, and we want to build a code for arbitrary k, n, and q. So we present our simplified version of EMC recursively on k, with the code for $k = 2$ described above as our base case.

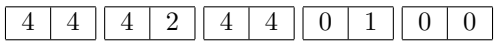

Figure 2.1: An example of the base case of EMC, where $n = 10, q = 5$. The ten cells are broken into five blocks of two cells each. The two active blocks are the second and fourth blocks. The second block is assigned to bit 0; we can tell because the left cell is in a state greater than that of the right cell. We know $v_0 = 0$ because the active cell is in state 2, which is even. The fourth block is assigned to bit 1; we can tell because the right cell is in a state greater than that of the left cell. We know $v_1 = 1$ because the active cell is in state 1, which is odd.

First, we introduce a key term used in the paper by Yaakobi, et al. We say a block *stores* a set of bits S if only flips of bits in S can cause the states of the cells in the block to increase. (Note that this does not imply that every flip of every bit in S will cause the block to be updated.) Now, suppose we have a code for $k = 2^{i-1}$, where the cells are broken into blocks of size 2^{i-1}, and each block stores bits $0, \ldots, (2^{i-2} - 1)$ or $2^{i-2}, \ldots, (2^{i-1} - 1)$. Then we construct a code for $k = 2^i$ (we assume, as Yaakobi et al. assume, that k is a power of 2) as follows:

Break the cells into blocks of size 2^i, and break each block into two subblocks of size 2^{i-1}. Use these subblocks just as you would in the code for $k = 2^{i-1}$, storing bits $2^{i-1}, \ldots, 2^i - 1$ as if they were bits $0, \ldots, (2^{i-1} - 1)$, with two caveats:

1. Each block stores either bits $0, \ldots, (2^{i-1} - 1)$ or bits $2^{i-1}, \ldots, (2^i - 1)$. For example, if one subblock stores bits $0, \ldots, (2^{i-2} - 1)$, then its neighboring subblock can store bits $0, \ldots, (2^{i-2} - 1)$ or $2^{i-2}, \ldots, (2^{i-1} - 1)$, but not bits in $2^{i-1}, \ldots, (2^{i-1} + 2^{i-2} - 1)$ or $(2^{i-1} + 2^{i-2}), \ldots, (2^i - 1)$.
2. If a block stores bits $0, \ldots, (2^{i-1} - 1)$ (resp. $2^{i-1}, \ldots, (2^i - 1)$), then the left (resp. right) subblock must have weight greater than the right (resp. left) subblock. If writing to a block would violate this rule, then start a new block, instead.

Because of rule 2, we must differentiate between *writeable* subblocks and *non-writeable* subblocks of an active block, where writeable subblocks can be written to without violating rule 2, but non-writeable subblocks have been frozen until the weight of the neighboring subblock increases.

To decode, we find whether each block stores bits $0, \ldots, (2^{i-1} - 1)$ or bits $2^{i-1}, \ldots, (2^i - 1)$ by comparing the weights of the subblocks, and then we decode each subblock according to the code for $k = 2^{i-1}$. When a bit is stored in more than one block at a time (which can happen because of rule 2), we XOR together the values of the bit from the different blocks — i.e. add (mod 2) the number of times (mod 2) the bit has been flipped as recorded in each of the active blocks — to find

the current value of the bit.

Examples

Suppose $k = 8$ and $q = 5$, and consider the following block:

1	4	4	3	1	0	0	0

The left subblock (| 1 | 4 | 4 | 3 |) has weight 12 and the right subblock (| 1 | 0 | 0 | 0 |) has weight 1, so since $12 > 1$, the block must store bits $0, ..., 3$. The left subblock of the left subblock (| 1 | 4 |) has weight 5 and the right subblock of the left subblock (| 4 | 3 |)has weight 7, so since $7 > 5$, the left subblock must store bits 2 and 3. Since $4 > 1$, the left subblock of the left subblock stores bit 3, and since $3 < 4$, the right subblock of the left subblock stores bit 2. Since 1 and 3 are odd, this block will contribute a 1 to the value of each of these bits, which will be XOR-ed with the values of these bits as stored in other blocks. Similarly, the right subblock stores bits 0 and 1, and the left subblock of the right subblock stores bit 0, which has current value 1. The right subblock of the right subblock is empty; it can store bit 0 or bit 1.

Here is an example of a block where the left subblock is active but unwriteable:

2	0	0	0	1	0	0	2

Since the weight of the right subblock is greater than the weight of the left subblock, we know this block stores bits $4, ..., 7$, so the right subblock must always have weight greater than the left subblock. If any cell in the left subblock increases value, then the weights of the two subblocks will become equal, which is forbidden. So if bit 4 or bit 5 is flipped next, the flip will have to be recorded in a new block (note that here, the left subblock records bits 4 and 5, and the right subblock records bits 6 and 7). Then the values of bits 4 and 5 as encoded by this block will be XOR-ed with the value of these bits as encoded in the new block to find the true current value of those bits.

EMC: A More Efficient Version

The code presented in the paper by Yaakobi et al. is like the code described above, with two differences:

1. They use a more complicated, more efficient base case, where $k = 4$ and there are at most 6 active blocks at a time, but the number of unused levels in these blocks is bounded.

2. They store bits $0 \ldots (\frac{k}{2} - 1)$ using the cells from left to right, and bits $\frac{k}{2} \ldots (k-1)$ using the cells from right to left. This is equivalent to storing two sets of $\frac{k}{2}$ bits in two separate arrays that grow until they meet. This allows them to use blocks of size $\frac{k}{2}$.

To determine the deficiency of this code, Yaakobi et al. count the maximum number of unused levels in the writeable and unwriteable subblocks, and come up with a recursion which they solve to prove that their code has deficiency $(\frac{7q}{16} - \frac{13}{48})k^2 - (q + \frac{1}{2})k + \frac{7}{3}$. However, their proof relies on a false assumption about the number of active blocks; we have contacted them about this error and are currently waiting for a reply. [1]

Implementation

In this section, we discuss our implementation of EMC. Below we discuss the merits of implementation and what we learned; see Appendix 2 for the source code of our implementation.

Why Implement?

In the field of floating codes, implementation provides insight into how a code could be implemented on a chip; for example, how much the deficiency would have to increase for the code to be efficiently executable, and how much time a read or write operation would take. Because the construction by Yaakobi et al. is so important, new, and complicated (the first version was significantly more complicated than the code presented above), we decided to implement the code. In the case of this particular code, implementation had an added benefit: while implementing the code, we discovered three errors, one of which was quite serious. The first error was a neglected corner case that, when fixed, increased the leading coefficient of the deficiency by a factor of 1.5. The second error was a fundamental problem in the recursion that required a rewriting of the encoding and decoding rules, again increasing the leading coefficient by a small factor. The code described above is from the third version of the paper, reflecting these changes in response to our correspondence. Upon completing our implementation of the third version of the code, we discovered another error, having to do with the maximum number of active blocks, which caused them again to rewrite their encoding rules.

[1] Shortly before submitting this thesis, we received a reply to our latest correspondence; the authors have rewritten the encoding and decoding rules to bound the number of active blocks, proving deficiency $\frac{3}{4}(q-1)k^2 - \frac{7}{2}(q-1)k + 1$ for their new version of EMC. Thus, we do not know for sure what the deficiency of the code as described above is, but it remains an important and influential code.

However, we received the fourth version of their paper too late to change our presentation of their code here (see Footnote 1).

Time vs. Space

Floating codes are designed to minimize deficiency, without regard to encoding and decoding speed. However, encoding and decoding times also matter, and might be improved by setting aside memory for useful data structures. Because this trade-off between time and space is so dependent on specifics of hardware, we will continue the tradition of not taking time into account when designing and analyzing floating codes. However, in this section we will discuss briefly the trade-off between time and space we came across while implementing EMC.

During our implementation, it quickly became apparent that, since $n >> k$, the rate limiting step in encoding and in decoding was walking through all of the full blocks before finding an active block. We cut down on this time by storing the locations of the k active blocks, which required setting aside $O(\frac{kn}{q})$ cells as pointers: k sets of $\frac{n}{q}$ cells to record k numbers less than n which only increase. In other words, to make our implementation of this code efficient, we had to make our deficiency again dependent on n. This is a problem likely to come up in any floating code.

Fortunately, this issue does not necessarily arise in hardware implementation, because of the parallelization capacity which does not exist in software. Instead of having to walk through all of the blocks, or to store k numbers of size up to n, it could be possible to sum all the blocks at the same time, finding which blocks are active in time $O(k)$ without setting aside extra cells. Parallelizing at the cell level could lead to read and write times as low as $O(1)$. Another possibility is in an on-chip look up table. However, comparison of these options is beyond the scope of this thesis.

Conclusions

The paper by Yaakobi, et al. was not just an important paper because it presented a code with such a low deficiency; the authors changed both the language and the standard for describing what makes a code "good." Deficiency is a much more precise measure of optimality than, for example, asymptotic optimality as defined by Jiang et al., and EMC showed how close to optimal a code could be, using almost every level possible. On a more practical level, the idea introduced by Jiang et al. and developed by Yaakobi et al. of breaking cells into blocks, storing groups of bits in different blocks, and limiting the number of active blocks proved particularly influential in our work, as we demonstrate in Chapter 3.

Chapter 3

A Simpler Code, Improved Asymptotic Performance

In this chapter, we present two original codes, based on the EMC described in Chapter 2. In Section 3.1 we describe a simple code with deficiency $O(k^2q)$; in Section 3.2 we define a new procedure, "marking off;" and in Section 3.3 we present a secondary code with deficiency dependent on n. In Section 3.4 we combine all of these elements into a new, iterative code with deficiency $O(kq\log^2 k)$, and in Section 3.5, we discuss possible extensions of these codes.

3.1 Basic Mod-Based Code

Preliminaries

We will assume for this code that k divides n, and that q is odd. As in the code by Yaakobi et al., we will imagine our n cells as divided into blocks of k cells, and we will use the terms *empty*, *full*, and *active* as defined in Section 2.2.2. However, in our code, every active block will be assigned to only one bit. When we use the phrase "mod k," we mean that we are temporarily supposing that the last cell and the first cell of a given block are adjacent, and that the first cell is to the right of the last.

Intuitively, each bit b fills up a block left to right (mod k), starting at the b^{th} cell. Because cell $b-1 \pmod k$ is always the rightmost (mod k) empty cell, or the only active cell in an otherwise full block, it is always possible to tell that this block is recording bit b, and to decode accordingly.

For example, if $k = 8$ and $q = 5$, the block

4	2	0	0	4	4	4	4

stores bit 4; it began by writing to cell 4 (recall that we are counting from 0, not 1), then continued to cells 5, 6, and 7, and then wrapped around to cells 0 and 1. We know $v_4 = 0$ because 2, the state of the only active cell, is even. We formalize these rules as follows.

Encoding

Suppose bit b is flipped. We start by finding the active block that is assigned to it, or if no active block is assigned to it, we assign the left-most empty block to it. Call this block A. There are three cases:

1. A is empty. In this case, increase the state of the b^{th} cell by one.

2. A has an active cell. In this case, increase the state of the active cell by one.

3. A does not have an active cell, and there is a unique empty cell with a left neighbor (mod k) that is full. In this case, increase the state of that empty cell by one.

To show that these rules are well-defined, we must show that these cases are the only cases that can arise.

Lemma 3.1.1. *Every active block is made up of a contiguous (mod k) set of full cells followed by one or zero active cells, followed by a contiguous (mod k) set of empty cells, where either set can be empty.*

Proof. By induction. Initially, all blocks are empty, so the condition is trivially satisfied. Now, suppose that all active blocks are in the state described, bit b is flipped, and block A is assigned to bit b. If A was empty before b was flipped, then A now consists of a contiguous (mod k) set of empty cells and one active cell, satisfying the required description (where the set of full cells is the empty set). If A had an active cell in a state less than $q - 2$, then all empty cells remain empty, the active cell remains active, and all full cells remain full. If A had an active cell in state $q - 2$, then

this cell becomes full, so the block now has no active cells, but its full cells are still contiguous (mod k) because the formerly active cell is adjacent to the set of full cells, as are its empty cells, which have not changed. If A had no active cells but at least one full cell and one empty cell, then the left-most (mod k) empty cell becomes the single active cell, keeping both the empty and full cells contiguous. □

Corollary 3.1.2. *The encoding rules are well defined.*

Decoding

To decode, we look at each active block. If there are any empty cells in the block, there must be a unique empty cell with a non-empty cell to the immediate right; this is cell $b-1 \pmod k$, where the block is assigned to bit b. Otherwise, there is a unique active cell and all other cells are full; in this case, the active cell is the cell $b-1 \pmod k$. If there is no active cell in the block, then $v_b = 0$. Otherwise, $v_b = x \pmod 2$, where x is the weight of the active cell. If there is no block assigned to a bit, then $v_b = 0$.

The decoding rules correctly identify the bit assigned to each active block because the blocks are filled from left to right (mod k), starting at the b^{th} cell. They correctly decode the value of each bit because q is odd and so full cells are not relevant.

Deficiency Analysis

Proposition 3.1.1. This code has deficiency $k^2(q-1) - kq + 1$.

Proof. The code ends when a bit b is flipped and no block is assigned to b, but there are no empty blocks. In this case, there are at most $k-1$ active blocks (since each active block must be assigned to a bit that is not bit b) and all other blocks must be full. In these $k-1$ active blocks, at most $k-1$ cells are empty, and at most $q-2$ levels are unused by the active cell. So the deficiency is $(k-1)((k-1)(q-1)+(q-2)) = k^2(q-1) - kq + 1$. □

3.2 Marking Off

The deficiency of a code is made up of two parts: empty cells and unused levels in active cells. Sometimes, it is possible to use sequences of cell states called *markers* to *mark off* the empty cells for future use in a *secondary code*. The important property of markers is that they are never used in the secondary code.

For example, after encoding with Basic Mod-Based Code (BMBC), the empty cells appear in at most $k-1$ contiguous sets of cells, and if we do not use the cell state sequences $(1,1)$ and $(2,2)$ in our secondary code, we can use $k-1$ sets of them to mark off all but $4(k-1)$ of the left over empty cells. So for each block with more than four empty cells, we can set the first two (mod k) of these cells to $(1,1)$, and the last two (mod k) to $(2,2)$, thereby marking off the empty cells in between. To avoid any ambiguity, we set all cells which are not markers and have not been marked off to state $q-1$, except for the right-most (mod k) of these cells, which we set to $q-1$ or $q-2$ to preserve the parity of the bit that was recorded in that block. If a block which records bit b has fewer than five empty cells, then we fill the block, except for cell $b-1 \pmod{k}$, which we set either to $q-1$ or $q-2$. (This method of preserving parity assumes that $q-2>2 \Rightarrow q \geq 5$. In the case that this assumption does not hold, there are other ways to preserve the information without affecting the asymptotic performance of the code, for example by reserving one block to record the values of the bits.) We can then treat the cells as one row of blank cells to be used in our secondary code, as long as the sequences $(1,1)$ and $(2,2)$, our markers, never arise.

For example, suppose $k=8, q=5$, and we have the following three active blocks after running BMBC:

0	0	4	4	2	0	0	0	0	4	4	4	4	1	0	0	0	0	0	0	1	0	0	0

(The first block is assigned to bit 2, the second block is assigned to bit 1, and the third block is assigned to bit 4; bit 2 is currently set to 0, bit 1 is set to 1, and bit 4 is set to 1.) Now we mark off, to get

2	2	4	4	4	1	1	0	3	4	4	4	4	4	4	4	0	0	2	2	3	1	1	0

Note that since there were not enough empty cells in the second block to mark off, we simply filled the whole block, leaving cell $b-1=0$ in state $q-2$ to preserve the parity. We also filled the active cells of the other blocks up to parity; if we hadn't, then the last block would now be

0	0	2	2	1	1	1	0

and it would be unclear whether the last 1 was part of the marker, or a cell that had been marked off and partially used in the secondary code.

Now, we have marked off four cells, which we can treat as one empty row of cells:

0	0	0	0

and we can write to this new block. For example, if according to our secondary code, we should set these cells to states

1	2	4	1

then our original three blocks would be updated like this:

2	2	4	4	3	1	1	1	4	4	4	4	4	4	4	3	4	1	2	2	3	1	1	2

To read this, we find the left and right markers in the first and third blocks, and read the numbers stored in between (mod k). Note that three ones in a row is unambiguous, because we filled all the cells to the left of the left marker when we marked off.

When analyzing a code that will be marked off, it is important to distinguish between levels lost — that is, levels in cells which will not be used at any later point — and cells that are marked off. Because of this, we introduce the *marked deficiency* of an encoding scheme, which is an ordered pair (a, b), where a is the maximum number of levels lost, including all the levels of the cells used as markers, and b is the maximum number of cells which could be marked off. If an encoding scheme has marked deficiency (a, b), then it has deficiency $a + b(q - 1)$. In the case of BMBC, we lose all the levels leftover from all active cells, and we lose at most 4 cells for each contiguous set of empty cells — at most $4(k-1)(q-1) + (k-1)(q-2) = 5kq - 6k - 5q + 6$ levels — and we save at most $\frac{(k^2(q-1)-kq+1)-(5kq-6k-5q+6)}{q-1} = k^2 - 6k + 5$ cells for future use. So what we have shown is:

Proposition 3.2.1. BMBC has marked deficiency $(5kq - 6k - 5q + 6, k^2 - 6k + 5)$.

3.3 A Secondary Code

We now introduce our secondary code, with marked deficiency $(2kq\lceil\log_{q-2} k\rceil + o(kq\lceil\log_{q-2} k\rceil), \frac{n}{2} + o(n))$. Here, we assume $q-2 \leq k$, and $\lceil\log_{q-2} k\rceil$ is the number of cells needed to encode a number between 1 and k without using 1's or 2's; if $k < q - 2$, then substitute 1 for $\lceil\log_{q-2} k\rceil$ wherever it appears. We also assume that $\lfloor\frac{n}{2k}\rfloor > \lceil\log_{q-2} k\rceil$.

Encoding

We arrange the cells into a row of $2k$ blocks of size $\lfloor\frac{n}{2k}\rfloor$ or $\lceil\frac{n}{2k}\rceil$, with the larger blocks to the left of the smaller blocks, using as many small blocks as possible. Each block records exactly one bit, and from each block, we set aside the last $\lceil\log_{q-2} k\rceil$ cells as *indexing cells* to record which bit is stored in that block.

Each time bit b flips, we find the block which records bit b, or if such a block does not exist then we record the number b in the last $\lceil\log_{q-2} k\rceil$ cells of the leftmost empty block, where indexing cells are never in states 1 or 2 (to avoid ambiguity with markers). We then increase the state of the

active cell in that block by one, or if there is no active cell, then we increase the state of the leftmost empty cell. We stop when a bit flips but no block records it, and there are no empty blocks.

Decoding

To decode, we divide the cells into $2k$ blocks as in the encoding scheme, and read which bit is recorded in each active block from the last $\lceil \log_{q-2} k \rceil$ cells. We then decode the value of the bit recorded in each active block — if there is an active cell that is not an indexing cell, then the parity of the state of this cell is the value of the bit; otherwise, the bit has value 0 — and any bit with no active block is in state 0.

Marking Off and Analysis

When the code ends, there are at most $k-1$ active blocks and at most one active non-indexing cell per block. This means that there are at most $(k-1)(\lceil \frac{n}{2k} \rceil - \lceil \log_{q-2} k) \rceil) < \frac{n}{2}$ empty non-indexing cells, and they are in at most $k-1$ contiguous rows.

When marking off after the secondary code, we can use a more efficient marking scheme than the scheme we used to mark off after BMBC. Because each block is written left to right and we know that the row of empty cells ends with the cell $\lceil \log_{q-2} k \rceil$ from the right of the block, we only need a left marker. Our left marker can simply be the leftmost non-full cell in the block, which we will ensure is not a 1 or a 2. So if there are no active cells to the left of the indexing cells when our code ends, we increase the level of the next empty cell to be $q-2$ or $q-3$ (now assuming $q \geq 6$; again, there are other ways to preserve this information if $q \not\geq 6$), and this cell functions as our marker and preserves parity at the same time. Otherwise, we raise the level of the single active cell, now our marker, to be $q-2$ or $q-3$. This means we lose at most $(k-1)(q-1)$ levels to the cells used as markers and the active cells we fill, since if there is an active cell then it is also the marker for that block. In addition, we lose $2k(q-1)\lceil \log_{q-2} k \rceil$ levels to indexing, and we mark off at most $(k-1)(\lceil \frac{n}{2k} \rceil - \lceil \log_{q-2} k \rceil - 1)$ cells, giving us marked deficiency $((k-1)(q-1)+2k(q-1)\lceil \log_{q-2} k \rceil, (k-1)(\lceil \frac{n}{2k} \rceil - \lceil \log_{q-2} k) \rceil - 1)) = (2kq \log_{q-2} k + o(kq \log_{q-2} k), \frac{n}{2} + o(n))$.

Note that throughout this process, we never have two consecutive ones or two consecutive twos, and that as long as the same is true of the code we use on the marked off cells, no other cell combinations need to be avoided by this code.

3.4 Combining and Iterating: an Iterative Mod-Based Code

Now, we can construct an encoding scheme called an Iterative Mod-Based Code (IMBC) with deficiency $O(kq(\log_2 k)(\log_{q-2} k))$. We start by running BMBC and marking off, as described in Section 3.2, and are left with fewer than k^2 cells, losing at most $5kq - 6k - 5q + 6$ levels. We then run our secondary code and mark off, giving us fewer than $\frac{k^2}{2}$ cells, losing at most $(k-1)(q-1) + 2k(q-1)\lceil \log_{q_2} k \rceil$ levels. We then continue to run the secondary code and mark off, until we have fewer than $k(\lceil \log_q k \rceil + 1)$ cells. This will take fewer than $2\log_2 k$ iterations, and at each iteration we lose at most $(k-1)(q-1) + 2k(q-1)\lceil \log_{q_2} k \rceil$ levels, giving us an overall deficiency of $(5kq - 6k - 5q + 6) + 2\log_2 k((k-1)(q-1) + 2k(q-1)\lceil \log_{q_2} k \rceil) = 4kq \log_2 k \lceil \log_{q-2} k \rceil + o(kq(\log_2 k)(\log_{q-2} k))$.

3.5 Error Correction and Other Extensions

IMBC is asymptotically better than the previous state of the art in floating codes. However, the simplicity of BMBC has several benefits that are worth discussing. To begin with, it is easy to understand and easy to implement, and still has a deficiency independent of n. But it is also an interesting code in the context of error-correction and rank-modulation, with minor adjustments. Here, we will present two examples; further exploration of these areas is a topic for future research.

BMBC-ECC

When programming flash memory, the cost of programming a cell to a state higher than the state intended ("overshooting") is high, as mentioned in Chapter 1. To that end, error-correcting codes that focus on identifying and correcting the errors caused by overshooting are important. BMBC can easily be modified into an error-correcting code of that type, called BMBC, Error Correcting Code (BMBC-ECC). In BMBC-ECC, we use blocks of twice the size and instead of having one active cell at a time per block, we have two, which we use alternately. So when a block is first assigned to bit b, we raise the state of cell $2b$ to one, but the next time bit b is flipped, we raise the state of cell $2b+1$, then cell $2b$, then cell $2b+1$, etc. Once these two cells are full, we record in cells $2b+2$ and $2b+3 \pmod k$. The usage of blocks, and the decoding of which bit is stored in which block follows exactly as in BMBC. Details of intra-block encoding, decoding, and error-correction are as follows.

We call a block *legitimate* if the following are true:

1. The block is made up of a contiguous (mod k) set of full cells followed by zero, one, or two active cells, followed by a contiguous (mod k) set of empty cells.

2. If there is one active cell, it is in state 1 (if it is an even cell) or $q-2$ (if it is an odd cell). If there are two active cells c_i and c_{i+1} in states s_i and s_{i+1}, then $s_i - s_{i+1} \in \{0, 1\}$ and i is even. If there are no active cells, then the number of full cells is even.

3. The leftmost (mod k) full cell is an even cell.

To update, if there are no active cells, or one active cell in state one, then we increase the state of the left-most (mod k) empty cell. If there is one active cell in state $q-2$, then we increase the state of that cell to $q-1$. If there are two active cells and the left active cell is in a state one greater than the right active cell, then we increase the state of the right active cell. If there are two active cells in the same state, then we increase the state of the left active cell. All these increases are only one level. To decode, if there are zero active cells, or two active cells in the same state, then the bit has value 0. If there is one active cell, or two active cells in different states, then the bit has value 1.

Lemma 3.5.1. *If a block is legitimate and then updated once without error, then it will remain legitimate. If a block is legitimate and there is an overshoot, then the block will not remain legitimate.*

Proof. First, when we say, "an overshoot of x states," we mean an overshoot where the state of a cell is raised by x states, instead of one.

Case 1: Before the update, there are zero active cells. The update will increase the state of the leftmost (mod k) empty cell, an even cell. If there is no overshoot, then there will be one active cell (an even cell) in state one, and the block will remain legitimate. If there is an overshoot of fewer than $q-2$ states, then the cell will be in a state greater than one but less than $q-2$. Since this cell will be the only active cell, the block will not be legitimate. If there is an overshoot of $q-2$ states, then the block will have one active cell in state $q-2$, but it will be an even cell, and therefore not legitimate. If there is an overshoot of $q-1$ states, then there will be only full and empty cells, but there will be an odd number of full cells, so we will know the block is not legitimate.

Case 2: Before the update, there is one active cell in state 1. The update will increase the state of the empty cell to the immediate right of the active cell. If there is no overshoot, then there will be two active cells in state one, so the block will be legitimate. If there is an overshoot of fewer than $q-1$ states, then there will be two active cells, but the active cell to the right will be in a state greater than the active cell to the left, so the block will not be legitimate.

If there is an overshoot of $q-1$ states, then either the full cells will not be contiguous, or the left-most full cell will not be an even cell.

Case 3: Before the update, there is one active cell in state $q-2$. The next update will increase the state of this cell to full; the block will remain legitimate, and there can be no overshoot.

Case 4: Before the update, there are two active cells in the same state, which is greater than one. If there is no overshoot, the state of the left active block will be increased by one, and the block will remain legitimate. If there is an overshoot that does not make the left cell full, then there will be two active cells whose states differ by more than one, and the block will not be legitimate. If there is an overshoot that makes the left cell full, then there will be only one active cell — an odd cell. If it is in state one, the block will not be legitimate because the active cell is odd. Otherwise, the active cell will not be in state one or $q-2$, also making the block not legitimate. (Note that if the right active cell was in state $q-2$, then so was the left active cell, so no overshoot was possible.)

Case 5: Before the update, there are two active cells and the left active cell is in a state one greater than the right active cell. The update will increase the state of the right active cell. If there is no overshoot, it will bring the active cells to the same state, and the block will remain legitimate. If there is an overshoot that does not fill the right cell, then the right active cell will be in a state greater than the left active cell, resulting in a block that is not legitimate. If there is an overshoot that fills the right cell, then either the full cells will not be contiguous, or the left-most (mod k) full cell will not be even.

□

Proposition 3.5.1. If there is an overshoot, it can be detected and corrected.

Proof. That we can detect an overshoot follows from the lemma. In fact, the proof of the lemma also tells us how to detect which cell was overshot. We can see this by checking that each potential non-legitimate block configuration is achievable only from an overshoot in one particular cell.

- If the left-most full cell is odd, then this cell was overshot.
- If there is one active cell which is even and in a state between 2 and $q-3$, then it was overshot.

- If there is one active cell which is odd and in a state between 2 and $q-3$, then the cell to its left was overshot.
- If there is one active cell which is odd and in state one, then the cell to its left was overshot.
- If there is one active cell which is even and in state $q-2$, then it was overshot.
- If the full cells are not contiguous, then the one full cell whose removal would make the full cells contiguous was overshot.
- If there is an odd number of full cells but no active cells, and the left-most full cell is even, then the right-most full cell was overshot.
- If there are two active cells and the left active cell is in a state more than one greater than the state of the right active cell, then the left active cell was overshot.
- If there are two active cells and the right active cell is in a state greater than the state of the left active cell, then the right active cell was overshot.

Since these are the only block configurations that are not legitimate that will arise (by the proof of the lemma), we have shown that it is always possible to detect which cell was overshot. □

To correct the error, first note that when a bit flips from 0 to 1, the state of an even cell will increase, and when a bit switches from 1 to 0, the state of an odd cell will increase. So by knowing which cell was overshot, we know in which state the bit is, and we can increase the states of appropriate cells to get a legitimate block that records the correct value of the bit.

BMBC-RM

Another way of dealing with the problem of overshooting is not to specify target states when programming cells. In a new paper ([15]), Jiang et al. describe a new encoding technique that takes into account only the relative values of the cells, decoding based on the permutations. Here, we briefly sketch a similar code, Basic Mod-Based Code, Rank Modulation (BMBC-RM), which is nearly identical to BMBC-ECC. We again use two cells at a time, but we code only based on whether the left or right active cell is higher. If the left active cell is higher, or if there is only one active cell, then the bit has value 1; if the right active cell is higher or there are no active cells, then the bit has value 0. There is no need to have q specific states; we simply set a cell detectably higher than its neighbor at each bit flip.

Other Extensions

BMBC-ECC and BMBC-RM are two examples of how BMBC can be extended to address the problem of overshooting, but there are other potential problems to be addressed, and the flexibility of BMBC makes it a promising starting point to think about these other problems. Within each block, all that needs to be preserved is the knowledge of which cells are empty and which bit is encoded in the rightmost (mod k) non-empty cells. Only the empty cells determine which bit is encoded in a block; only the rightmost non-empty cells encode the value of the bit. We showed above how useful it could be to change the way the right-most non-empty cells store the value of the bit. As another example of the flexibility of BMBC, consider the problem of cells losing charge slowly over time [4]. As long as the full cells do not become totally empty, and as long as the right-most (mod k) cells preserve the correct value of the bit, this creates no problem. Other improvements on this code are a promising area for future work.

Chapter 4

Average-Case Floating Codes

Worst-case floating codes aim to maximize the guaranteed number of updates between reset operations. While it is valuable to establish lower bounds on performance, what will actually determine the lifespan of a flash cell is how often reset operations occur on average. For example, suppose for simplicity that a given chip of flash memory wears out after 1,000 reset operations. If a code is used that guarantees ten updates per reset operation, but never achieves more than ten updates, then the chip will wear out after exactly 10,000 updates. If, however, a code is used in which with 50% probability there will be 20 updates in between two erasures, and with 50% probability there will be 6, then the expected life of the chip will be 13,000 updates, and the probability of wearing out after fewer than 10,000 updates is less than 10^{-84}. So with high probability, the second code will result in longer life for a chip, despite inferior worst-case performance.

Average-case codes must assume an underlying probabilistic model for how the variable vector will change. The main advantage of worst-case codes is that their lower bounds are guaranteed for any underlying probabilistic behavior. However, if information could be collected on how variable vectors tend to change, then codes could be optimized for these behaviors, potentially improving their performance significantly. The main contribution of this chapter is a framework for analyzing floating codes in the average case. The constructions of this chapter focus on simple underlying models; future work should involve codes for probabilistic models constructed from real data.

In Section 4.1, we introduce the average-case model for floating codes. In Section 4.2, we present two codes which highlight the difference between average-case and worst-case floating codes. In Section 4.3, we present some principles behind the construction of average-case codes. In Section 4.4, we give an example of how to extend constructions for small values of n to larger values of n.

In Section 4.5, we discuss the subset-flip model of floating codes, where any subset of the bits can flip in a given timestep. This chapter, with the exception of Proposition 4.5.1, is original exposition of work done in collaboration with Flavio Chierichetti, Zhenming Liu, and Michael Mitzenmacher which has been pubished [5]. Proposition 4.5.1 is a contribution of this thesis.

4.1 The Average-Case Model

Let us suppose that the transitions among variable vectors can be modeled by a Markov chain with transition matrix T_v, where the probability of switching from variable vector x to variable vector y at a given timestep is the $(x, y)^{th}$ entry of T_v ($T_v(x, y)$). Our previous codes have focused on a model where only one bit flips at a time; this corresponds to the case in which $T_v(x, y) = 0$ if x and y differ by more than one bit. The Markov chain on the variable vectors, together with the update function U and the decode function D, induces a Markov chain on the cell state vectors with transition matrix T_c, where for cell state vectors s and t, $T_c(s, t) = T_v(D(s), D(t))$ if $D(s)$ and $D(t)$ differ by one bit b and $U(s, b) = t$, and 0 otherwise. (See Section 1.3 for a definition of U and D.) A second model, where a variable vector can transition to any other variable vector instead of only to vectors that differ by one bit, is discussed in Section 4.5. While more detailed models will eventually be called for, these two models provide a good basis on which to build.

Suppose t and s are cell state vectors, with $t = (t_1, \ldots, t_n)$ and $s = (s_1, \ldots, s_n)$. If $t_i \geq s_i$ for all i, we say $t \geq s$; note that $t \geq s$ if and only if a transition from s to t can occur without a reset operation. Let π_s be the stationary probability of cell state vector s for our Markov chain; recall that this gives the fraction of time spent, on average, in state s. At an intuitive level, a good code is one in which at any given time, the probability that the next update will require a reset operation is small. The probability that the next update will require a reset operation is the same as the sum over all states s of the probability of being in state s times the probability that an update from state s will require a reset operation. Based on this reasoning, we define the *cost* of a code to be $\sum_{t \not\geq s} \pi_s p_{s,t}$. If the cost of a code is c, then $\frac{1}{c}$ is the average number of updates between reset operations, and $n(q-1) - \frac{1}{c}$ is the average-case deficiency. Our goal is to minimize cost.

There are two ways to calculate cost. The first is to find T_v, construct T_c from T_v, determine the left principal eigenvector v of T_c, normalize v to sum to one, build a matrix T_c' by setting all entries (s, t) of T_c to be 0 if $s \leq t$, multiply vT_c', and sum the resulting vector. This gives a precise answer, but quickly becomes inconvenient for large values of n or q, since T_c is a $q^n \times q^n$ matrix. An alternative method of finding cost is simulation.

4.2 A Worst-Case and an Average-Case Code for $k = 2$, $n = 2$, $q = 4$

Having defined average case codes, we will now show that codes that work well in the worst case do not always work well in the average case, and conversely that codes that work well in the average case do not always work well in the worst case. Jiang et al. [13] show that no code for $k = 2, n = 2$ can guarantee more than $q - 1 + \lfloor \frac{q-1}{2} \rfloor$ updates (see Section 2.1.2). Here, we present an example of an optimal worst-case code for $k = 2, n = 2, q = 4$, based on a code by Yaakobi et al. [19]; we will call it 2-dimensional worst case, or 2DWC. (In fact, Yaakobi et al. present a generalization of this code for arbitrary n and odd q which we describe in Section 4.4, but for now we will describe our adaptation for $q = 4, n = 2$.)

2DWC

We present the decode function D of this code in an array, where the $(i, j)^{th}$ entry is $D(i, j)$; the upper left corner corresponds to cell state vector (0,0), and upper right corner corresponds to cell state vector (0,3). The update function greedily updates to the closest cell state vector that decodes to the appropriate bit string. For example the sequence of variable vectors 00-01-00-10 is encoded into cell state vectors 00-01-02-12.

00	01	00	01
10	11	10	10
00	01	00	11
10	11	01	00

Yaakobi et al. prove that their generalization of this code is optimal; inspection is enough to verify this claim for the case we have described, since four updates are guaranteed before a reset.

How does this code perform on average? First, we must define an underlying probabilistic model for the possible changes of a variable vector. As an example, we will assume a simple model: that bit 0 is the next to flip with probability $p_0 = 0.7$ and bit 1 is the next to flip with probability $p_1 = 0.3$. This gives us the following transition matrix T_v:

	00	01	10	11
00	0	0.3	0.7	0
01	0.3	0	0	0.7
10	0.7	0	0	0.3
11	0	0.7	0.3	0

T_v induces a much larger transition matrix on the cell state vector, T_c (see Figure 4.1). The principal left eigenvector of this matrix is $v = (0, 0.045, 0.013, 0.004, 0.094, 0.143, 0.052, 0.016, 0.066, 0.120, 0.073, 0.029, 0.046, 0.098, 0.119, 0.080)$. To find the cost, we create a second matrix T_c' by replacing all the entries (s, t) of T_c with 0, except for those where $t \not\geq s$. We then multiply vT_c', and sum. Our result, the cost, is 0.2227; for this model, a reset occurs every 4.4901 updates, on average.

2DGC+

2DV was designed for the worst case; we now introduce a second code, designed for the average case, called 2-dimensional Gray Code Plus (2DGC+). It is described by the following array, again letting the $(i, j)^{th}$ entry of the array be $D(i, j)$, and decoding greedily (for more details on the design of 2DGC+, see Section 4.3).

00	01	11	10
10	00	01	11
11	10	00	01
01	11	10	11

Using the strategy of 4.2.1, we can find the cost of 2DGC+. T_v has not changed, but the induced T_c has (see Figure 4.2). The principal eigenvector of this new matrix is $v = (0.082, 0.094, 0.066, 0.020, 0.094, 0.094, 0.074, 0.058, 0.028, 0.074, 0.074, 0.063, 0.020, 0.036, 0.063, 0.063)$, and the new cost is 0.1874; a reset occurs every 5.3362 updates, on average.

Note that 2DGC+ does not guarantee four updates before a reset. For example, if the variable vector were to be updated 00-01-11-10-00, then the fourth update, from 10 to 00, would require a reset. So in the worst case, 2DGC+ performs more poorly than 2DV. But according to this particular probabilistic model, we have shown that 2DGC+ performs better than 2DV on average by a significant margin. It may seem that our results are based only on our choice of probabilities, but a sensitivity analysis using $p_0 \in \{0.1, \ldots, 0.9\}$ and $p_1 = 1 - p_0$ yields similar results (see Figure 4.3).

4.3 Some Principles of Average-Case Construction

Why does 2DGC+ perform so well? There are a few principles underlying its construction that give intuition that it should perform well in the average case. To begin with, we differentiate between *inner* cell state vectors and *edge* cell state vectors; $c = (c_0, \ldots, c_{n-1})$ is an inner cell state vector

	00	01	02	03	10	11	12	13	20	21	22	23	30	31	32	33
00	0	0.3	0	0	0.7	0	0	0	0	0	0	0	0	0	0	0
01	0	0	0.3	0	0	0.7	0	0	0	0	0	0	0	0	0	0
02	0	0	0	0.3	0	0	0.7	0	0	0	0	0	0	0	0	0
03	0	0	0	0	0	0	0	0	0	0	0	0.7	0	0	0	0.3
10	0	0	0	0	0	0.3	0	0	0.7	0	0	0	0	0	0	0
11	0	0	0	0	0	0	0.3	0	0	0.7	0	0	0	0	0	0
12	0	0	0	0	0	0	0	0.3	0	0	0.7	0	0	0	0	0
13	0	0	0	0	0	0	0	0	0	0	0	0.3	0	0	0	0.7
20	0	0	0	0	0	0	0	0	0	0.3	0	0	0.7	0	0	0
21	0	0	0	0	0	0	0	0	0	0	0.3	0	0	0.7	0	0
22	0	0	0	0	0	0	0	0	0	0	0	0.3	0	0	0.7	0
23	0	0.7	0	0	0.3	0	0	0	0	0	0	0	0	0	0	0
30	0	0	0	0	0	0	0	0	0	0	0	0	0	0.3	0	0.7
31	0	0	0	0	0.3	0	0	0	0	0	0	0	0	0	0.7	0
32	0	0	0	0	0	0.7	0	0	0	0	0	0	0	0	0	0.3
33	0	0.3	0	0	0.7	0	0	0	0	0	0	0	0	0	0	0

Figure 4.1: T_c for 2DV. For lack of space, we denote the cell state vector $\{c_0, c_1\}$ by c_0c_1. So the probability that the next cell state vector will be (0,3), given that it is currently (0,2), is 0.3.

	00	01	02	03	10	11	12	13	20	21	22	23	30	31	32	33
00	0	0.3	0	0	0.7	0	0	0	0	0	0	0	0	0	0	0
01	0	0	0.7	0	0	0.3	0	0	0	0	0	0	0	0	0	0
02	0	0	0	0.3	0	0	0.7	0	0	0	0	0	0	0	0	0
03	0.7	0	0	0	0	0	0	0.3	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0.7	0	0	0.3	0	0	0	0	0	0	0
11	0	0	0	0	0	0	0.3	0	0	0.7	0	0	0	0	0	0
12	0	0	0	0	0	0	0	0.7	0	0	0.3	0	0	0	0	0
13	0	0	0	0	0.3	0	0	0	0	0	0	0.7	0	0	0	0
20	0	0	0	0	0	0	0	0	0	0.3	0	0	0.7	0	0	0
21	0	0	0	0	0	0	0	0	0	0	0.7	0	0	0.3	0	0
22	0	0	0	0	0	0	0	0	0	0	0	0.3	0	0	0.7	0
23	0.3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0.7
30	0.3	0	0	0	0	0	0	0	0	0	0	0	0	0.7	0	0
31	0	0.7	0	0	0	0	0	0	0	0	0	0	0	0	0.3	0
32	0.7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0.3
33	0	0.7	0	0	0.3	0	0	0	0	0	0	0	0	0	0	0

Figure 4.2: T_c for 2DGC+.

p_0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
2DV	0.2128	0.2112	0.2101	0.2101	0.2119	0.2161	0.2227	0.2314	0.2409
2DGC+	0.1763	0.1831	0.1874	0.1897	0.1905	0.1897	0.1874	0.1831	0.1763

Figure 4.3: Average cost per update for 2DV and 2DGC+ at different values of p_0.

if $c_i < q - 1$ for all i, and an edge cell state vector otherwise. Each inner cell state vector has n *neighboring* cell state vectors that it can reach by increasing the state of one cell by one. So if $k \leq n$, then an inner cell state vector should be able to accommodate any bit flip by increasing one cell's state by one. This principle holds for 2DGC+.

When $k = n$, there exists for each edge cell state vector a bit b such that when b flips, a cell must increase by more than one level (by the pigeonhole principle). Thus, it is always better to be at an inner cell state vector than an edge cell state vector. A second principle of average-case codes is to maximize the amount of time spent in inner cell state vectors by keeping cell levels as near to each other as possible. We call this *sticking to the diagonal*. In the case of $n = 2$, this principle is achieved through the use of Gray codes as building blocks. 2DGC+ comes from an earlier code called 2-Dimensional Gray Code (2DGC), which is based on the Gray code sequence 00, 01, 11, 10 and is represented in the following diagram:

00 01 11 10
10 00 01 11
11 10 00 01
01 11 10 00

(The only difference between 2DGC and 2DGC+ is the lower-right corner.) 2DGC was built on the assumption that $p_0 \neq p_1$. Without loss of generality, we will say $p_0 > p_1$. The idea is that when bit 0 is flipped repeatedly, instead of one cell filling up while the other remains empty — leading to early arrival at an edge cell state vector — the cells will alternate. This corresponds to the diagonal path given by:

00 01 11 10
10 **00** 01 11
11 **10** **00** 01
01 11 **10** **00**

2DGC can be extended to arbitrary q by tiling. For example, 2DGC for $q = 8$ is given by the following array:

00	01	11	10	00	01	11	10
10	00	01	11	10	00	01	11
11	10	00	01	11	10	00	01
01	11	10	00	01	11	10	00
00	01	11	10	00	01	11	10
10	00	01	11	10	00	01	11
11	10	00	01	11	10	00	01
01	11	10	00	01	11	10	00

It is possible to use a Chernoff bound to show that as q gets large, 2DGC has deficiency $O(\sqrt{q \log q})$ with probability $1 - \frac{1}{\text{poly}(q)}$; this is significantly better than lower bound $O(q)$ on worst-case performance of a code for $n = 2$.

However, 2DGC will never use the cell state vector in the upper left corner, because any time the variable vector is updated to 00, the lower right hand corner can be used instead. This leads us to our third principle, which is to use as many cell state vectors as possible. 2DGC has a redundancy, and replacing the 00 in the lower right hand corner with 11 (getting 2DGC+) we can eliminate this redundancy, improving performance slightly.

4.4 Extending Average-Case Codes to Large n and k

These principles have been applied to get codes for other small values of n and k, and some of these codes have been used as building blocks in larger codes or to improve on the average-case performance of optimal worst-case codes [5]. In this section, we give an example of how 2DGC can be used as a building block for a code for $k = 2$, odd q, and arbitrary n, based on the worst-case optimal code by Yaakobi et al. [19]. We have a generalization of 2DGC to large values of q (see Section 4.3); now we must introduce a similar generalization of 2DWC.

The generalization of 2DWC for $n = 2$ and odd q is given by Yaakobi et al. To decode, when $c_i < q-1$ for both i, $v_i = c_i \pmod 2$, and when there is an i such $c_i = q-1$, then $v_0 = \lceil \frac{c_j \pmod 4}{2} \rceil$ and $v_1 = c_j \pmod 2$, where $j \neq i$. To update, change the cell state vector to the closest available cell state vector that decodes to the correct bit string.

Now we can describe the code 2DWC-n for arbitrary n and q by Yaakobi et al. Picture the cells as arranged in a row, and encode as follows:

1. If, before encoding, there are more than two non-full cells,
   - If bit 0 flips, increase the state of the left-most non-full cell.
   - If bit 1 flips, increase the state of the right-most non-full cell.
2. If, after encoding with rule 1 or before encoding, there are two or fewer non-full cells, encode according to the general version of 2DWC.

For example, if $q = 5, n = 10$, and the cell state vector is | 4 | 4 | 4 | 4 | 2 | 0 | 3 | 4 | 4 | 4 |, then $v_0 = 0$ and $v_1 = 1$ because the left-most active cell is in state 2, which is even, and the right-most active cell is in state 3, which is odd. If the variable vector changes as follows: 01-11-10-00-10, then the cell state vector will change as follows:

4	4	4	4	2	0	3	4	4	4
4	4	4	4	3	0	3	4	4	4
4	4	4	4	3	0	4	4	4	4
4	4	4	4	4	0	4	4	4	4
4	4	4	4	4	2	4	4	4	4

A naive way to adapt this code for the average case would be to store the bits similarly until there are only two cells left that are not in state $q - 1$, and then to store the two bits in these two cells according to 2DGC+. However, it would be possible that when the encoding switched to 2DGC+, the states of the two cells would not be similar; that is, we would not be near the diagonal. So instead, we adapt this construction by storing each bit in two cells at a time, instead of one. A pair of cells can store a single bit by "walking on the diagonal" as the bit flips; i.e. as the bit flips the two cells will be updated as follows: $00 - 10 - 11 - 21 - 22 - \ldots - (q-1)(q-2) - (q-1)(q-1)$; here $D(c_0, c_1) = (c_0 + c_1) \mod 2$. We can split our cells into pairs and use the pairs of cells to store bit 0 from left to right and bit 1 from right to left as above. So when there are only two cells left which are not in state $q - 1$, the difference in their states will be at most one, and we can switch to using 2DGC+. For example, if the cell state vector is | 4 | 4 | 4 | 4 | 3 | 2 | 2 | 2 | 4 | 4 |, then the bit 0 is encoded by | 3 | 2 |, so $v_0 = 1$, and bit 1 is encoded by | 2 | 2 |, so $v_1 = 0$. Now, if bit 0 flips repeatedly, then the cell state vector will change as follows:

4	4	4	4	3	3	2	2	4	4
4	4	4	4	4	3	2	2	4	4
4	4	4	4	4	4	2	2	4	4

4	4	4	4	4	4	3	2	4	4

Now cells 6 and 7 store both bits, using 2DGC+.

4.5 The Subset Flip Model

In Section 1.3, we defined a worst-case code as an update function $U : \{0, \ldots, q-1\}^n \times \{0,\ldots,(k-1)\} \to \{0, \ldots, q-1\}^n \cup \{\perp\}$, together with a decode function $D : \{0, \ldots, q-1\}^n \to \{0,1\}^k$, subject to the constraint that if $U(s, b) = t$, then $D(s) \oplus D(t) = e_b$, where e_b is the bit string of all zeros except for a one in the b^{th} position. This refers to the *bit-flip* model, where only one bit can flip in a given timestep; codes for the bit-flip model are called *bit-flip codes*. There is a second interesting model, the *subset-flip* model, where at each timestep the whole bit string can be rewritten. The subset-flip model, as the name implies, is equivalent to the model where any subset of the bits can flip in a given time step, and a *subset-flip code* is defined by an update function $U : \{0, \ldots, q-1\}^n \times \{0,1\}^k \to \{0, \ldots, q-1\}^n \cup \{\perp\}$ and a decode function $D : \{0, \ldots, q-1\}^n \to \{0,1\}^k$. There is a natural correspondence between subsets of $[k]$ and bit strings from $[2]^k$, where i is in the subset if and only if the i^{th} element of the bit string is 1, so when we refer to an element of $\{0,1\}^k$, we are referring both to a bit string and to a subset of $[k]$. With this language, we have the property that U and D define a valid code if and only if $U(s, b) = t$ implies that $D(s) \oplus D(t) = b$. Below, we demonstrate a relationship between these two models and state a lower bound on the worst-case deficiency of a subset-flip code. We then use random codes to prove the existence of codes that do significantly better on average than is possible in the worst case, under the assumption that at each timestep a new variable vector is chosen uniformly at random.

Proposition 4.5.1. Any bit-flip code for $K = 2^k$ variables can be converted into a subset-flip code for k variables, with the same values of n and q and the same deficiency.

Proof. First, let $[a]$ denote the set $\{0, \ldots, a-1\}$, and let $[a]^b$ denote the set of strings of length b with elements from $[a]$. Now, let $f : [K] \to [2]^k$ be the bijection given by the enumeration of bit strings in lexicographic order, and $\pi : [2]^K \to [2]^k$ be the function given by $\pi(x_1 x_2 \ldots x_K) = \bigoplus_{x_i=1} f(i)$. Note that $\pi(X) \oplus \pi(Y) = \pi(X \oplus Y)$.

Now, suppose we have a bit-flip code C for n, q, and K variables given by $U : [q]^n \times [K] \to [q]^n \cup \{\perp\}$ and $D : [q]^n \to [2]^K$. Define $U' : [q]^n \times [2]^k \to [q]^n \cup \{\perp\}$ and $D' : [q]^n \to [2]^k$ by $U'(s, x) = U(s, f^{-1}(x))$ and $D'(s) = \pi(D(s))$. Then $U'(s, x) = t$ implies that $U(s, f^{-1}(x)) = t$, and so

$$\begin{aligned} D'(s) \oplus D'(t) &= \pi(D(s)) \oplus \pi(D(t)) \\ &= \pi(D(s) \oplus D(t)) \\ &= \pi(e_{f^{-1}(x)}) \\ &= x \end{aligned}$$

These two functions satisfy the requisite property, and so we have defined a new subset-flip code C'. Now, we need to show that C' has the same deficiency as C. Beginning from the bit string of all zeros, a sequence of bit flips $b_1 \ldots, b_m$ will be encoded via U into exactly the same cell state vectors as the sequence of subset flips $f(b_1), \ldots, f(b_m)$ via U'. So C guarantees t writes if and only if C' guarantees t writes, and therefore the two codes have the same deficiency.

□

In Chapter 3, we described a worst-case bit-flip code with deficiency $kq(\log k)^2$; the above proposition then gives us a worst-case subset-flip code with deficiency $2^k qk^2$. The lower bound on the deficiency of a subset-flip code is $2^k q$ ([5]), so we are close to achieving the lower bound.

In contrast, we know very little about bounds and constructions for average-case subset-flip codes. Below, we give a probabilistic argument that there exist codes with average-case deficiencies under 2^k, based on a proposition by Chierichetti et al. ([5]). However, there is not yet a known lower bound on the average-case deficiency of a subset-flip code, so we cannot even know whether an average-case deficiency of 2^k is "good," even though it is a factor of q better than the lower bound on worst-case deficiencies.

We begin by defining a family of random codes called *Least* codes. Least codes are generated by choosing, for each cell state vector s, a value for $D(s)$ uniformly at random from the set of variable vectors. We then define $U(s, x)$ as follows: if there is a cell state vector t which has weight one greater than the weight of s, and such that $D(t) = D(s) \oplus x$, then we set $U(s, x) = t$. If there is more than one such t, then we break the tie by choosing the cell state vector which is "closest to the diagonal"; i.e. the cell state vector where the difference between the highest cell state and the lowest cell state is minimized. If there is no such t, then we increase the state of the cell with the lowest state and repeat the search.

The following proposition is due to Chierichetti et al. ([5]); the proof is beyond the scope of this thesis.

Proposition 4.5.2. For any fixed sequence of subset flips, the expected deficiency of a Least code is $2^k + o(1)$ as n increases, provided that $2^k \log(nq) \in o(\sqrt{n})$.

Corollary 4.5.1. *There exist subset-flip codes with average-case deficiencies below* 2^k*, provided that* $2^k \log(nq) \in o(\sqrt{n})$.

Proof. By the proposition, the expected deficiency over all sequences of subset flips and all Least codes is $2^k + o(1)$. Letting $D(C)$ denote expected deficiency of a fixed Least code C over all sequences of subset fips and $E[D(C)]$ denote the expected value of $D(C)$ over all Least codes, $E[D(C)]$ is also $2^k + o(1)$. Since it is possible to construct Least codes with expected deficiency worse than 2^k (for example by assigning all cell state vectors to the same variable vector), this means that there are Least codes with expected deficiency lower than 2^k. □

The Least codes are based heavily on the principles outlined in Section 4.3; Proposition 4.5.1 shows that these principles are quite powerful.

Chapter 5

Conclusions

The work done in this thesis leaves several avenues open for future work. In Chapter 3, we introduced a simple and flexible floating code with deficiency $O(k^2q)$, and an iterative floating code with deficiency $O(kq \log^2 k)$. The current known lower bound on the deficiency of a floating code is $O(kq)$, so there is still progress needed to reduce this gap. There is also room for improvement adapting the first code to the problems of error correction, and, more generally, coming up with codes that address the idiosyncrasies and errors of flash memory other than just the increase/decrease asymmetry, but which still have low deficiencies.

There is also work to be done improving the average-case performance of these codes. First, it would be helpful to have codes for general values of n, k, and q under simple Markov models. But we also need a better understanding of how variable vectors change before we can apply these codes to flash memory.

While much progress has been made in various aspects of the problem of coding for flash memory, there are also many avenues for future research to explore in each of these areas, and in combining them into a code that is flexible and efficient.

Chapter 6

Appendices

6.1 Flash Memory

Each cell on a chip of flash memory is a floating gate transistor. A floating gate transistor is made of a body of semiconductive material ("substrate"), with a layer of insulating material ("oxide") on top. On top of the oxide there is a layer of metal, called the control gate, and in the middle of the oxide is another layer of conductive material, called the floating gate. The layer of oxide between the substrate and the floating gate is called tunnel oxide, and the layer of oxide between the control gate and the floating gate is called interpoly oxide. There are also two terminals at the interface between the oxide and the substrate, each connected to highly doped regions in the substrate (regions in the substrate that act like conductors). These are called the source and the drain (See Figure 6.1)

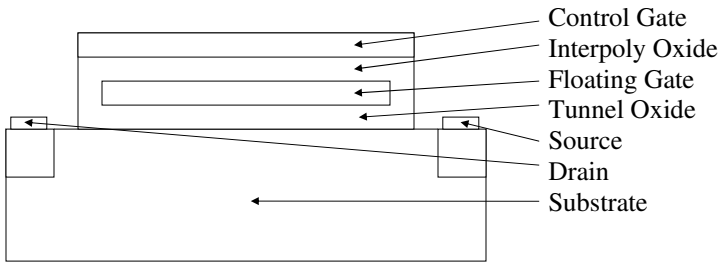

Figure 6.1: The layout of a flash cell [3].

When enough voltage is applied to the control gate, the voltage repels the positive holes (small areas of positive charge) in the substrate, causing a channel of electrons to form between the source and the drain. The amount of voltage needed for this channel to form is the *threshold voltage* V_t. When the floating gate stores negative charge, more voltage is needed to form the channel, so the threshold voltage increases to a new value V_t'. By applying a certain voltage V that is in between V_t and V_t', it is possible to determine whether the floating gate is charged or not; if a channel is formed at voltage V then there is no charge stored on the gate, and if no channel is formed then there is charge stored on the gate [3]. For binary flash, we say the cell is in state 0 in the first case, and is in state 1 in the second case. (Not everyone follows this convention; in some descriptions, the first case corresponds to a one and the second case corresponds to a zero.)

To write to a cell, then, it is necessary to inject electrons onto the floating gate, which is surrounded by insulating material. This is done in one of two ways: channel hot electron (CHE) injection, where voltage is applied to the control gate at the same time that a strong electric field is created between the source and the drain so that some high-energy electrons pass through the tunnel oxide to the floating gate; or Fowler-Nordheim (FN) tunneling, where a voltage is applied to the control gate that is strong enough to create a quantum tunnel to the floating gate, charging the floating gate. FN tunneling is also used to erase ("reset") a cell, but when used as an erasing mechanism, it can be used only on many cells at a time, is time consuming, and can cause charge leakage when used too frequently. Because it is so costly to reset cells, it is desirable to have as many write operations as possible in between reset operations [2].

In the newer multilevel cells, rather than just two levels of charge on a floating gate (off or on), there are q levels, where q can range from 2 to 256. This requires the application of more than one voltage to the control gate to determine which level the floating gate is in, but the real difficulties arise with accurate writing. Because there are more than two potential levels of charge on the floating gate, the margin of error for each level is smaller, so much more precision is needed when injecting electrons. Also, because cells can only be reset in large sections, the cost of injecting more charge than intended is large. As a result, the current method of injecting electrons involves multiple iterations, carefully approaching the desired level from below. This process is time-consuming and costly [15].

6.2 Implementation of EMC

We implemented EMC in MATLAB. Below are the .m files we wrote, which we used to simulate encoding and decoding.

```
function [num_flips] = simulate(probs,n,k,q)
%simulates bits flipping according to a probability distribution
%given by probs, storing the information in a cell state vector with
%parameters n, k, and q. It returns the number of times a bit flipped
%before a reset operation was needed. It uses one_dir to update each time a
%bit is flipped.

%Since we are recording from right to left and from left to right, we will
%have two different CSV's, which will grow until their sum is bigger than
%2*n/(k-1). They start out with one block each.
CSV = cell(1,2);
Active_Blocks = cell(1,2);
for i = 1:2
    CSV{i} = cell(1);
    CSV{i}{1} = zeros(1, k/2);
    Active_Blocks{i} = 1;
end

%probs is a vector of length k that sums to one, where the i-th element is
%the probability that bit i will be the next to flip. We transform probs to
%simulate bits flipping.

probs = [0 cumsum(probs)];

num_flips = 0;

%Flip a bit at random, determine whether to record it in the right or the
%left, and then update CSV and Active_Blocks using one_dir.
b = find(probs<rand,1,'last');
i = (b > k/2) + 1;
[CSV{i},Active_Blocks{i}] = one_dir(b-(i-1)*k/2,CSV{i},Active_Blocks{i},n,k/2,q);
```

```
%Continue to flip bits, stopping when the two sides meet.
while length(CSV{1})+length(CSV{2}) < 2*n/k
    num_flips = num_flips+1;
    b = find(probs<rand,1,'last');
    i = (b > k/2) + 1;
    [CSV{i},Active_Blocks{i}] = one_dir(b-(i-1)*k/2,CSV{i},Active_Blocks{i},n,k/2,q);
end



function  [CSV,Active_Blocks] = one_dir(b,CSV,Active_Blocks,n,k,q)
%One_dir updates the cell state vector, the array of active blocks when bit
%b flips once. It does this by checking each active block, starting with
%the one with the smallest index, to see if it stores bit b.

num_blocks = length(Active_Blocks);

%tried keeps track of which active blocks have already been checked
tried = zeros(1,num_blocks);

%The search is done in two passes: first checking only active subblocks, and
%then checking both active and empty subblocks. This avoids the
%problem of starting a new subblock in a block with a low index when there
%is an active subblock storing the appropriate bit already. counter keeps
%track of which pass is happening.
counter = 0;

%i keeps track of how many active blocks have been checked.
i = 0;

while i < num_blocks + 1
    %Find the minimum active block that hasn't yet been tried. Adding
    %tried*n assures that the minimum will not be a block that has already
    %been tried. index tells where the minimum block appears in the list
    %Active_Blocks.
    [which_block,index] = min(Active_Blocks+tried*n);
```

```
Block = CSV{which_block};
tried(index) = 1;

%Now we iterate through the subblocks of different levels. Agenda keeps
%track of which subblock we are on: the top row keeps track of starting
%cell of the subblock, and the bottom row keeps track of the size. We
%begin with the whole subblock of size k.
agenda = [1;k];

while ~isempty(agenda)  %while agenda is not empty
    start_cell = agenda(1,1);
    range = agenda(2,1);

    %If the subblock is actually a cell, increase the value of that
    %cell by one. if the block that was just updated is now full, take
    %it off the list of active blocks.
    if range==1
        CSV{which_block}(start_cell) = CSV{which_block}(start_cell) + 1;
        if sum(CSV{which_block})==k*(q-1)
            Active_Blocks = Active_Blocks(Active_Blocks~=which_block);
        end
        return
    end

    %If the subblock is not a cell, compare the sums of the two halves
    %of the subblock.
    sums(1)= sum(Block(start_cell:start_cell+range/2-1));
    sums(2)= sum(Block(start_cell+range/2:start_cell+range-1));

    %Figure out which half would have to be bigger for this subblock to
    %record bit b.
    which_half = (mod(b-1,range) >= range/2)+1;

    %If the appropriate half is bigger
    if sums(which_half) > sums(3-which_half)
        %If the bigger half is not full, add it to the agenda.
```

```
            if sums(which_half) ~= range/2*(q-1)
                agenda = [agenda, [start_cell+(which_half-1)*range/2;range/2]];
            end
            %If the smaller half can be increased without changing which
            %half is bigger, or if the bigger half is full, add the smaller
            %half to the agenda.
            if sums(which_half)-sums(3-which_half) > 1 || sums(which_half) == range/2*(q-1)
                agenda = [agenda, [start_cell+(2-which_half)*range/2;range/2]];
            end

        %If this is the second pass, then if the whole subblock is empty,
        %add both halves to the agenda.
        elseif counter == 1 && sums(1)+sums(2)==0
           agenda = [agenda,[start_cell+range/2*(mod(b-1,range)>=range/2);range/2]];
        end

        %Now that you are done with this subblock, take it off the agenda.
        agenda = agenda(:,2:end);
    end

    %If you are done the first pass, then begin the second pass. If you are
    %done the second pass, then start the next empty block.
    if i == num_blocks-1 && counter == 0
        i = 0;
        counter = 1;
        tried = zeros(1,num_blocks);
    elseif i == num_blocks-1 && counter == 1
        tried = [tried 0];
        Active_Blocks = [Active_Blocks, length(CSV)+1];
        CSV = {CSV{1:end} 1:k};
        i = num_blocks;
    else
        i = i+1;
    end

end
```

```
function bitstring = decode(CSV,ActiveBlocks,k)
%This function takes a cell state vector in the form of an array of
%matrices (blocks), a matrix of the indices of the active blocks, and k. It
%returns the values of bits encoded in by this cell state vector.

bitstring = zeros(1,k);

num_blocks = length(ActiveBlocks);

%The algorithm does a DFS for each active block of the implicit tree of
%subblocks. To find which bits are stored in the cells at the leaves of the
%tree, a vector bitsum is kept which keeps track of the contributions of
%different subblocks -- ancestors of the leaf -- to the number of the bit
%stored in the leaf. For example, if the block stores bits k/2+1 ... k,
%then the root of the tree will contribute k/2 to bitsum. When the DFS
%reaches a leaf, the bit stored in that leaf is the sum of bitsum.

for i = 1:num_blocks
    which_block = ActiveBlocks(i);
    Block = CSV{which_block};
    bitsum = [];

    %As with the encoding function, agenda keeps track of which subblock is
    %being searched. Agenda(1,1) is the left-most cell of the subblock, and
    %range is the size of the subblock.
    agenda = [1;k];

    while ~isempty(agenda)
        start_cell = agenda(1,1);
        range = agenda(2,1);

        %Make sure bitsum is only as long as the depth of the node you are
        %at, so that only the contributions of the ancestors of a leaf will
```

```
        %be added in determining which bit is stored in the leaf.
        bitsum = bitsum(1:log(k/range)/log(2));

        %If you have reached a leaf (i.e. cell), add the appropriate value
        %to the bitstring, and move on.
        if range == 1
            bitstring(sum(bitsum)+1) = mod(bitstring(sum(bitsum)+1)+Block(start_cell),2);
            agenda = agenda(:,2:end);

        %If the subblock is not empty, determine whether it represents bits
        %1...range/2 or range/2+1...range, and add either 0 or range/2 to
        %bitsum, accordingly. Then add the two halves of the subblock to
        %the agenda.
        else
            sums(1)= sum(Block(start_cell:start_cell+range/2-1));
            sums(2)= sum(Block(start_cell+range/2:start_cell+range-1));
            if sums(1)+sums(2) > 0
                which_half = sums(2)>sums(1);
                bitsum = [bitsum which_half*range/2];
                agenda = agenda(:,2:end);
                agenda = [[start_cell, start_cell+range/2;range/2, range/2] agenda];
            else
                agenda = agenda(:,2:end);
            end
        end
    end
end
```

Bibliography

[1] R. Ahlswede and Z. Zhang. Coding for write-efficient memory. *Information and Computation*, pages 80–92, 1989.

[2] R. Bez, E. Camerlenghi, A. Modelli, and A. Visconti. Introduction to flash memory. *Proceedings of the IEEE*, 91(4):489–502, 2003.

[3] P. Cappelletti, C. Golla, P. Olivo, and E. Zanoni. *Flash Memories*. Kluwer Academic Publishers, 1999.

[4] Y. Cassuto, M. Schwartz, V. Bohossian, and J. Bruck. Codes for multi-level flash memories: correcting and asymmetric limited-magnitude errors. *Proc. IEEE International Symposium on Information Theory*, 2007.

[5] F. Chierichetti, H. Finucane, Z. Liu, and M. Mitzenmacher. Designing floating codes for expected performance. *Proc.* 46^{th} *Annual Allerton Conference on Communication, Control and Computing*, September 2008.

[6] A. Etengoff. Nand flash memory sales plummet. *IT Examiner*, March 2009.

[7] A. Fiat and A. Shamir. Generalized 'write-once' memories. *IEEE Transactions on Information Theory*, IT-30:470–480, 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 Transactions on Information Theory*, 45(1):308–313, 1999.

[9] F. Fu and R. W. Yeung. On the capacity and error-correcting codes of write-efficient memories. *IEEE Transactions on Information Theory*, 46(7):2299–2314, 2000.

[10] C. Heegard. On the capacity of permanent memory. *IEEE Transactions on Information Theory*, IT-31:34–42, 1985.

[11] C. Heegard and A. El Gamal. On the capacity of computer memory with defects. *IEEE Transactions on Information Theory*, IT-29:731–739, 1983.

[12] A. Jiang. On the generalization of error-correcting wom codes. *Proc. IEEE International Symposium on Information Theory*, pages 1391–1395, 2007.

[13] A. Jiang, V. Bohossian, and J. Bruck. Floating codes for joint information storage in wrte asymmetric memories. *Proc. IEEE International Symposium on Information Theory (ISIT)*, June 2007.

[14] A. Jiang, V. Bohossian, and J. Bruck. Joint coding for flash memory storage. *Proc. IEEE International Symposium on Information Theory (ISIT)*, July 2008.

[15] A. Jiang, R. Mateescu, M. Schwartz, and J. Bruck. Rank modulation for flash memories. *Proc. IEEE International Symposium on Information Theory*, 2008.

[16] R. L. Rivest and A. Shamir. How to reuse a 'write-once' memory. *Information and Control*, 55:227–231, 1984.

[17] G. Simonyi. On write-unidirectional memory codes. *IEEE Transactions on Information Theory*, 35(3):663–669, 1989.

[18] J. K. Wolf, A. D. Wyner, J. Ziv, and J. Korner. Coding for a write-once memory. *AT&T Bell Laboratories Technical Journal*, 63(6):1089–1112, 1984.

[19] E. Yaakobi, A. Vardy, P. Siegel, and J. Wolf. Multidimensional flash codes. *Proc.* 46^{th} *Annual Allerton Conference on Communication, Control and Computing*, 2008.