Trajectory Codes for Flash Memory

Anxiao Jiang, *Senior Member, IEEE*, Michael Langberg, *Member, IEEE*, Moshe Schwartz, *Senior Member, IEEE*, and Jehoshua Bruck, *Fellow, IEEE*

***Abstract*—A generalized rewriting model is defined for flash memory that represents stored data and permitted rewrite operations by a directed graph. This model is a generalization of previously introduced rewriting models of codes, including floating codes, write-once memory codes, and buffer codes. This model is used to design a new rewriting code for flash memories. The new code, referred to as trajectory code, allows stored data to be rewritten as many times as possible without block erasures. It is proved that the trajectory codes are asymptotically optimal for a wide range of scenarios. In addition, rewriting codes that use a randomized rewriting scheme are presented that obtain good performance with high probability for all possible rewrite sequences.**

***Index Terms*—Codes, flash memory, nonvolatile memory.**

I. Introduction

FLASH memory typically uses discrete states to store digital information. A flash memory consists of floating-gate cells, where a cell uses the charge it stores to represent data. The amount of charge stored in a cell can be quantized into q discrete levels in order to represent up to $\log_2 q$ bits. For $q = 2$, a cell is called a *single-level cell (SLC)*, and for $q > 2$, a cell is called a *multilevel cell (MLC)*. We call the q levels of a cell: level 0, level $1, \ldots,$ level $q-1$. The most common number of levels in flash cells using current technology is $q = 2, 4, 8$[22]. Using techniques such as rank modulation [17], virtual flash-memory cells with $n!$ levels may be created from n ordinary flash-memory cells.

The level of a cell can be increased by injecting a charge into the cell, and decreased by removing charge from the cell. Flash memories have the property that it is easy to increase a cell's level, and that it is hard and very costly to decrease a cell's level. This follows from the fact that flash-memory cells are organized as blocks, where every block typically has 10^5 to 10^6 cells. To decrease any cell's level, the entire block needs to be erased to remove the charge from all the cells of the block, and then be reprogrammed. Block erasures are slow, energy consuming, and significantly reduce the longevity of flash memories, because every block can sustain only 10^4 to 10^5 erasures with guaranteed quality [3]. Therefore, it is highly desirable to minimize the number of block erasures.

In general, the hardware state-transition constraints of a memory can be described by a simple directed graph, where the vertices represent the memory states and the directed edges represent the feasible state transitions [5], [8]. Different edges may have different costs [7]. Based on the constraints, an appropriate coding scheme is needed to represent the data so that the data can be rewritten efficiently.

There has been a significant amount of research on memories with hardware state-transition constraints, including the work by Kuznetsov and Tsybakov on coding for defective memories [19], and [10], [12]. Other examples include write-once memory (WOM) [23], write-unidirectional memory [24]–[26], and write-efficient memory [1], [7]. Among them, WOM is the most related to the flash-memory model studied in this paper.

Unlike the fixed hardware constraints, the way input data to be stored can change its value with each rewrite depends on the data-storage application and the used data structure. A *rewriting model* captures the way data can change its value. Several specific rewriting models have been studied in the past, including *WOM codes* [4], [5], [8], [21], [23], [27], *floating codes* [6], [15], [16], [20], [33], and *buffer codes* [2], [32]. In WOM codes, with each rewrite, the data can change from any value to any other value. In floating codes, k variables $v_1, v_2, \ldots, v_k$ are stored, and every rewrite can change only one variable's value. The rewriting model of floating codes can be used in many applications where different data items can be updated individually, such as the data in the tables of databases, in variable sets of programs, and in repeatedly edited files. In buffer codes, k data items are stored in a first-in-first-out queue, and every rewrite inserts a new data item into the queue and removes the oldest data item. All the above rewriting models can be generalized with a graph model we introduce in this work, which we call the *generalized rewriting model*.

WOM was studied by Rivest and Shamir [23]. In a WOM, a cell's state can change from 0 to 1 but not from 1 to 0. This model was later generalized with more cell states in [5] and [8]. WOM codes aim to maximize the number of times that the stored data can be rewritten. A number of WOM code constructions have been presented over the years, including the tabular codes and linear codes in [23], the linear codes in [5], the codes constructed

Manuscript received December 24, 2010; revised August 01, 2012; accepted November 29, 2012. Date of publication March 07, 2013; date of current version June 12, 2013. This work was supported in part by the National Science Foundation (NSF) CAREER Award CCF-0747415, in part by the NSF under Grant ECCS-0802107, in part by the Israel Science Foundation under Grant 480/08, in part by the Open University of Israel Research Fund under Grant 46109 and Grant 101163, in part by the Binational Science Foundation under Grant 2010075, and in part by the Caltech Lee Center for Advanced Networking. This paper was presented in part at the 2009 IEEE International Symposium on Information Theory.

A. Jiang is with the Department of Computer Science and Engineering, Texas A&M University, College Station, TX 77843-3112 USA (e-mail: ajiang@cse.tamu.edu).

M. Langberg is with the Computer Science Division, Open University of Israel, Raanana 43107, Israel (e-mail: mikel@openu.ac.il).

M. Schwartz is with the Department of Electrical and Computer Engineering, Ben-Gurion University, Beer Sheva 84105, Israel (e-mail: schwartz@ee.bgu.ac.il).

J. Bruck is with the Department of Electrical Engineering, California Institute of Technology, Pasadena, CA 91125 USA (e-mail: bruck@paradise.caltech.edu).

Communicated by A. J. van Wijngaarden, Associate Editor for Communications.

Digital Object Identifier 10.1109/TIT.2013.2251755

using projective geometries [21], and the coset codes in [4]. Results on the capacity of WOM have been presented in [8], [11], [23], and [27]. Furthermore, error-correcting WOM codes have been studied in [34]. In all the aforementioned works, the rewriting model assumes no constraints on the data.

With the increasing importance of flash memories, the flash-memory model was proposed and studied recently in [2], [14], and [15]. The rewriting schemes include floating codes [13]–[16] and buffer codes [2], [13]. Both types of codes use the joint coding of multiple variables for better rewriting capability. Multiple floating codes have been presented, including the code constructions in [15] and [16], the flash codes in [20] and [33], and the constructions based on Gray codes in [6]. The floating codes in [6] were optimized for the expected rewriting performance. Several recent papers on WOM codes consider new applications to flash codes and present improved code constructions [18], [28]–[31].

We summarize the parameters of the main asymptotically optimal constructions known so far. The early works on WOM codes consider n flash-memory cells, input symbols from an alphabet of size ℓ, and no restriction on the input stream. Proof of existence was given in [23], where a construction for $\ell = \Theta(1)$ was provided in [5]. In a few later works [15], [16], [20], [33], a new setting was considered, called *floating codes*, in which k variables, each from an alphabet of size ℓ, are stored in the flash memory. Furthermore, each rewrite operation can change only the content of a single variable. The constructions described in [15] and [16] require $k = \Theta(1)$, $\ell = \Theta(1)$, or $n = \Omega(k \log k)$, $\ell = \Theta(1)$, while those described in [20] and [33] require $n = \Omega(k^2)$, $\ell = \Theta(1)$. A model of floating codes that assumes a random rewrite sequence was studied in [6] and a construction requiring $k = \Theta(1)$, $\ell = 2$ was given. Another variation, called *buffer codes*, was described in [2] and [32], in which the k variables store a sliding window of values. In each rewrite, a single value is pushed into the window, and the oldest value is pushed out of the window. The constructions of [2] and [32] require $n = \Omega(k)$, $\ell = \Theta(1)$.

In this paper, we focus on flash memories, and our objective is to rewrite data as many times as possible between two block erasures. Note that between two block erasures, the cell levels can only increase due to the hardware constraints. Another main goal of this work is generalizing previous results by designing rewriting codes for general data-storage applications. We present a novel rewriting code, called the *trajectory code*, which is provably asymptotically optimal (up to constant factors) for a very wide range of scenarios. The trajectory code includes WOM codes, floating codes, and buffer codes as special cases.

We also study randomized rewriting codes and design codes that are optimized for the expected rewriting performance in terms of the expected number of rewrite operations the code supports. A rewriting code is called *robust* if its expected rewriting performance is asymptotically optimal (up to a factor of $(1 - \varepsilon)$) for *all* rewrite sequences. We present a randomized code construction that is robust.

Compared to existing codes, the codes in this paper not only work for a more general rewriting model, but also provide efficiently encodable and decodable asymptotically optimal performance for a wider range of cases. Thus, this paper substantially expands the known results on rewriting codes.

The rest of this paper is organized as follows. In Section II, we give a formal definition of a flash memory and a specification of the behavior of its cells. It is shown that the rewriting models available in the literature can be represented by a single graph model, referred to as the generalized rewriting model. Further definitions as well as some notation are introduced to provide a framework for the presentation of the design of the rewriting codes proposed in this paper. In Section III, a new rewriting code for the generalized rewriting model, the *trajectory code*, is presented and its optimality is proven. In Section IV, robust codes optimized for expected rewriting performance are presented. Section V presents concluding remarks and topics for further research.

II. Preliminaries

Let us first introduce a mathematical model for the hardware constraints of flash-memory cells. We shall assume throughout the paper we are given n flash memory cells, each with q discrete levels representing values from $\mathbb{Z}_q$. The hardware constraints are captured in the following flash-memory model.

Definition 1: The state of a flash memory with n q-level cells is specified by $\mathbf{c} = (c_0, c_1, \ldots, c_{n-1}) \in \mathbb{Z}_q^n$, where c_i is the level of the ith cell. The cells can change from state $\mathbf{c} = (c_0, c_1, \ldots, c_{n-1})$ to state $\mathbf{c}' = (c'_0, c'_1, \ldots, c'_{n-1})$, $\mathbf{c} \neq \mathbf{c}'$, if and only if $c'_i \geqslant c_i$ for all $0 \leqslant i \leqslant n-1$. In that case, we say that $\mathbf{c}'$ is *above* $\mathbf{c}$. □

We shall distinguish between the state of the memory and the stored data in the memory. A user may provide an input data value to be stored in the memory. At some later time, the user may provide a different data value to be stored, replacing the older value. This will be called a *rewrite* operation. The *rewriting model* defined below specifies constraints on the way input data values change during rewrite operations. Such constraints may be the result of certain applications or data structures employed by the user.

Definition 2: The stored data and the possible rewrite operations are represented by a simple directed graph

$$\mathcal{D} = (\mathcal{V}, \mathcal{E}).$$

The vertices $\mathcal{V}$ represent all the values that the data can take. There is a directed edge (u, v) from $u \in \mathcal{V}$ to $v \in \mathcal{V}$ (where $v \neq u$) iff a rewrite may change the stored data from value u to value v. The graph $\mathcal{D}$ is called the *data graph*, and its number of vertices—which corresponds to the data's alphabet size—is denoted by

$$L = |\mathcal{V}|.$$

□

Throughout this paper, we assume that the data graph is strongly connected, i.e., for any $v_1, v_2 \in \mathcal{V}$, there is a directed path from v_1 to v_2 in $\mathcal{D}$. A sequence of rewrite values is defined by $\mathbf{v} = (v_1, v_2 \ldots)$ such that the ith rewrite changes the stored data from v_{i-1} to v_i. It is assumed the initial stored value is

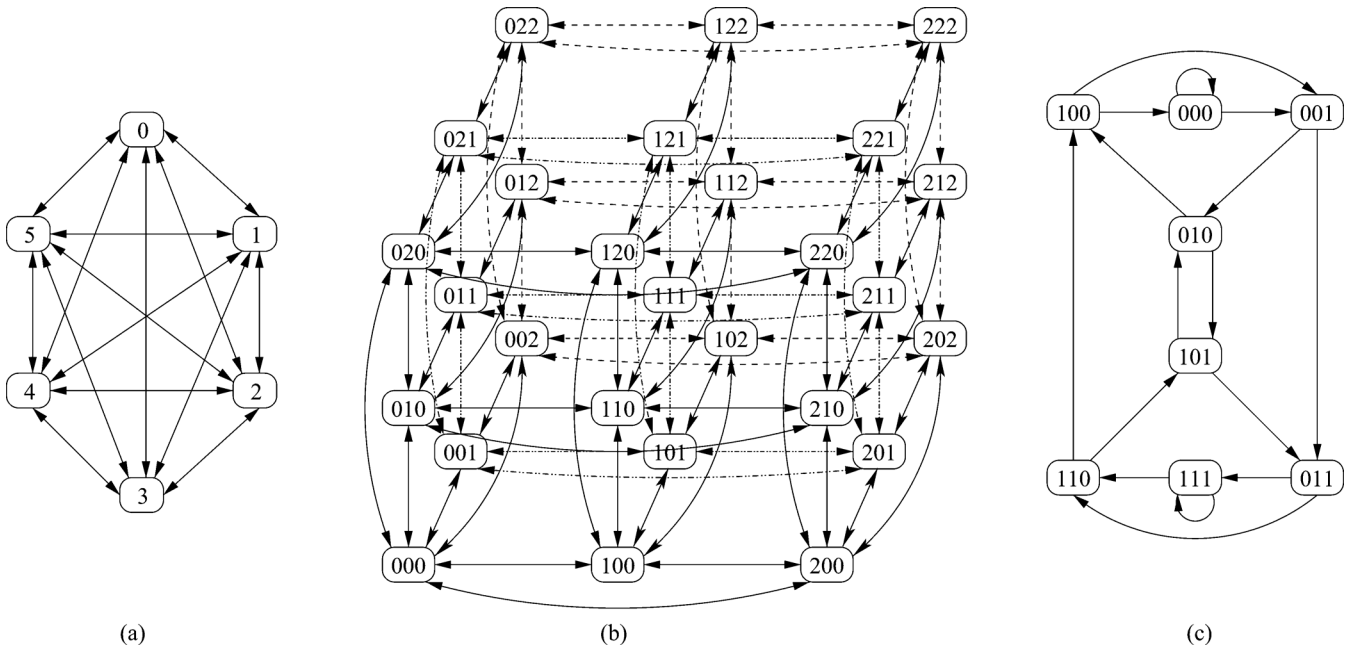

Fig. 1. (a) Complete data graph for a WOM code. (b) Hypercube data graph for a floating code. (c) De Bruijn data graph for a buffer code.

$v_0 = 0$, and that $(v_{i-1}, v_i) \in \mathcal{E}$ is a directed edge in the data graph $\mathcal{D}$.

Data graphs for different rewriting models are shown in Fig. 1. In the first example shown in Fig. 1, we see the data graph for a WOM code. The example shows data that have an alphabet of size 6. Since a rewrite in a WOM code can change the data from any value to any other value, the graph is complete. In Fig. 1(b), we see the data graph for a floating code. Here, $k = 3$ variables of alphabet size $\ell = 3$ are stored. Since in a floating code every rewrite can change exactly one variable's value, the graph is a generalized hypercube of regular degree $k(\ell - 1) = 6$ (for both out-degree and in-degree) in $k = 3$ dimensions. Finally, in Fig. 1(c), we see the data graph for a buffer code. In the example, $k = 3$ variables of alphabet size $\ell = 2$ are stored in a queue. Since in a buffer code every rewrite inserts a new variable into the queue and removes the oldest variable from the queue, the graph is a De Bruijn graph of degree $\ell = 2$.

With more data storage applications and data structures, the data graph can vary even further to capture a wide variety of data storage applications and data structures. This motivates us to study rewriting codes for the generalized rewriting model.

A rewriting code for flash memories can be formally defined as follows.

Definition 3: A rewriting code for flash memory with n q-level cells and a data graph $\mathcal{D} = (\mathcal{V}, \mathcal{E})$ has a *decoding function* F_d and an *update function* F_u. The decoding function

$$F_d : \mathbb{Z}_q^n \to \mathcal{V}$$

associates a cell state $s \in \mathbb{Z}_q^n$ with a vertex $F_d(s) \in \mathcal{V}$. The update function, representing a rewrite operation, and given by

$$F_u : \mathbb{Z}_q^n \times \mathcal{V} \to \mathbb{Z}_q^n$$

specifies that for a current state $s \in \mathbb{Z}_q^n$ and a new data value $v \in \mathcal{V}$, the rewriting code changes the cell state to $F_u(s, v)$. The following three properties must be satisfied:

1) $(F_d(s), v) \in \mathcal{E}$;
2) the cell-state vector $F_u(s, v)$ is above s;
3) $F_d(F_u(s, v)) = v$.

Note that if $F_d(s) = v$, we may set $F_u(s, v) = s$, which corresponds to the case where we do not need to change the stored data. Throughout the paper, we do not consider such a case as a rewrite operation. □

Given a rewriting code $\mathcal{C}$, we denote by $t(\mathcal{C})$ the maximal number of rewrite operations that $\mathcal{C}$ guarantees to support for all rewrite sequences. Thus, $t(\mathcal{C})$ is a worst case performance measure of the code. The code $\mathcal{C}$ is said to be *optimal* if $t(\mathcal{C})$ is maximized. In addition to this definition, if a probabilistic model for rewrite sequences is considered, the expected rewriting performance can be defined accordingly.

III. Trajectory Codes

We use the flash-memory model of Definition 1 and the generalized rewriting model of Definition 2 in the rest of this paper. We first present the construction of a novel code, referred to as *trajectory code*, and then show that this code's performance is asymptotically optimal.

A. Outline of the Constructions

Let n denote the number of q-level flash memory cells. As a first step, the n cells are partitioned into $d + 1$ subsets, called *registers*, of sizes $n_0, n_1, n_2, \ldots, n_d$, where

$$n = \sum_{i=0}^{d} n_i.$$

Let $\mathcal{D} = (\mathcal{V}, \mathcal{E})$ be the data graph, and denote by δ the maximal out-degree of the graph. We note that the out-degree of vertices may vary from 1 to δ.

The proposed encoding method uses the following basic scheme: we start by using register S_0, called the *anchor*, to record the value of the initial data $v_0 \in \mathcal{V}$.

For the next d rewrite operations, we use a differential scheme: denote by $v_1, \ldots, v_d \in \mathcal{V}$ the next d values of the rewritten data. In the ith rewrite, $1 \leqslant i \leqslant d$, we store in register S_i the identity of the edge $(v_{i-1}, v_i) \in \mathcal{E}$. We do not require a unique label for all edges globally, but rather require that *locally*, for each vertex in $\mathcal{V}$, its outgoing edges have unique labels from $\{1, \ldots, \delta\}$.

Intuitively, the first d rewrite operations are achieved by encoding the *trajectory* taken by the input sequence starting with the anchor data. After d such rewrite operations, we repeat the process by rewriting the next input from $\mathcal{V}$ in the anchor S_0, and then continuing with d edge labels in $S_1, \ldots, S_d$.

Let us assume a sequence of s rewrite operations have been stored thus far. To decode the last stored value, all we need to know is $s \bmod (d+1)$. This is easily achieved by using $\lceil t/(q-1) \rceil$ more cells (not specified in the previous $d+1$ registers), where t is the total number of rewrite operations that we would like to guarantee. For these $\lceil t/(q-1) \rceil$ cells, we employ a simple encoding scheme: in every rewrite operation, we arbitrarily choose one of those cells and raise its level by one. Thus, the sum of these cells' charge levels equals s.

The decoding process takes the value of the anchor S_0 and then follows $(s-1) \bmod (d+1)$ edges that are read consecutively from $S_1, S_2, \ldots$. Notice that this scheme is appealing in cases where the maximum out-degree of $\mathcal{D}$ is significantly lower than the size of the state space $|\mathcal{V}|$.

Note that each register S_i can be seen as a *smaller rewriting code* whose data graph is a *complete graph* of either L vertices (for S_0) or δ vertices (for $S_1, \ldots, S_d$). We let $d = 0$ if $\mathcal{D}$ is a complete graph, and describe how to set d when $\mathcal{D}$ is not a complete graph in Section III-C. The encoding used by each register is described in the next section.

B. Analysis for a Complete Data Graph

We now present an efficiently encodable and decodable code that enables us to store and rewrite symbols from an input alphabet $\mathcal{V}$ of size $L \geqslant 2$, when $\mathcal{D}$ is a complete graph. The information is stored in n q-level flash-memory cells.

We first state a scheme that allows approximately $nq/8$ rewrite operations in the case in which $2 \leqslant L \leqslant n$. We then extend it to hold for general L and n. We present the quality of our code constructions in terms of the number of possible rewrite operations using the asymptotic notation: $O(\cdot)$, $\Omega(\cdot)$, $\Theta(\cdot)$, $o(\cdot)$, and $\omega(\cdot)$, where n tends to infinity.

We shall now describe a code for small values of L. This code is essentially the one presented in [23].

Construction A: Let $2 \leqslant L \leqslant n$. This construction produces an efficiently encodable and decodable rewriting code $\mathcal{C}$ for a complete data graph $\mathcal{D}$ with L states, and flash memory with n cells with q levels each.

Let us first assume $n = L$. Denote the n cell levels by $\mathbf{c} = (c_0, c_1, \ldots, c_{L-1}) \in \mathbb{Z}_q^n$. Denote the data alphabet by $\mathcal{V} = \mathbb{Z}_L$. We first use only cell levels 0 and 1, and the data stored in the cells is

$$\sum_{i=1}^{L-1} i c_i \pmod{L}.$$

With each rewrite, we increase the minimum number of cell levels from 0 to 1 so that the new cell state represents the new data. (Clearly, c_0 remains unchanged as 0.) When the code can no longer support rewriting, we increase all cells (including c_0) from 0 to 1, and start using cell levels 1 and 2 to store data in the same way as above, except that the data stored in the cells uses the formula

$$\sum_{i=1}^{L-1} i(c_i - 1) \pmod{L}.$$

This process is repeated $q-1$ times. The general decoding function is therefore defined as

$$F_d(\mathbf{c}) = \sum_{i=1}^{L-1} i(c_i - c_0) \pmod{L}.$$

We now extend the above code to the case of $L \leqslant n$. We divide the n cells into $b = \lfloor n/L \rfloor$ groups of size L (some cells may remain unused). We first apply the code above to the first group of L cells. When that group has exhausted its rewrite capabilities, we apply the code above to the next group of L cells. The process is repeated until all groups are exhausted. □

We comment, in passing, that if L is odd, there is no need for cell c_0 in our construction. This slight improvement does not affect the asymptotics of our results.

For completeness, we describe the algorithm from [23] that finds the cells whose levels should be increased by 1 for the case of $n = L$. In this context, $\mathcal{V} = \mathbb{Z}_L$. Assume that the current state decodes to $v \in \mathcal{V}$, while we want to store $v' \in \mathcal{V}$, $v' \neq v$. We check whether cell $(v' - v) \bmod L$ can be increased by 1, i.e., we check whether $c_{(v'-v) \bmod L} = c_0$. If so, we increase it by 1. Otherwise, we go over all pairs of cells $1 \leqslant i < j \leqslant L-1$, $i + j \equiv v' - v \pmod{L}$, looking for pairs satisfying $c_i = c_j = c_0$. If we find such a pair, we increase cells i and j by 1. Otherwise, we declare failure. The complexity of this algorithm is $O(n^2)$ operations.

Theorem 4: Construction A guarantees that the number of rewrite operations of a code for n q-level cells and data alphabet size L is lower bounded by

$$t(\mathcal{C}) \geqslant n(q-1)/8 = \Omega(nq)$$

for any $2 \leqslant L \leqslant n$.

Proof: First assume $n = L$. When cell levels $j-1$ and j are used to store data (for $1 \leqslant j \leqslant q-1$), by the analysis in [23], even if only one or two cells increase their levels with each rewrite, at least $(L+4)/4$ rewrite operations can be supported. So the L cells can support at least

$$t(\mathcal{C}) \geqslant \frac{(L+4)(q-1)}{4} = \Omega(nq)$$

rewrite operations. Now let $n \geqslant L$. When $b = \lfloor n/L \rfloor$, it is easy to see that $bL \geqslant n/2$. The b groups of cells can guarantee

$$t(\mathcal{C}) \geqslant \frac{b(L+4)(q-1)}{4} \geqslant \frac{n(q-1)}{8} = \Omega(nq)$$

rewrite operations. ∎

We note that the denominator in the proof of Theorem 4 can be slightly improved using the results of [9].

We now consider the setting in which L is larger than n. The rewriting code we present reduces the general case to that of the case $n = L$ studied above. The majority of our analysis addresses the case in which $n < L \leqslant 2^{n/16}$. We start, however, by first considering the simple case in which $2^{n/16} \leqslant L \leqslant q^n$. We note that $2^{n/16}$ is used solely for the asymptotic analysis of the construction. For $L > q^n$, we observe that we cannot even guarantee a single rewrite operation.

Construction B: Consider n q-level flash-memory cells, a complete data graph $\mathcal{D}$ of size L, and a data alphabet $\mathcal{V} = \mathbb{Z}_L$. Denote the n cell levels by $\mathbf{c} = (c_0, c_1, \ldots, c_{n-1})$, where $c_i \in \mathbb{Z}_q$ is the level of the ith cell. The general decoding function is given by

$$F_d(\mathbf{c}) = \sum_{i=0}^{n-1} (c_i \bmod m) m^i \pmod{L}$$

where $1 \leqslant m \leqslant q$ is an integer parameter.

In the jth round of rewriting, we use levels jm to $(j+1)m-1$, for all $0 \leqslant j \leqslant \lfloor q/m \rfloor$, and with each rewrite, we represent $v \in \mathcal{V}$ by its n-character representation over an alphabet of size m. □

The following theorem is immediate.

Theorem 5: Let $\lambda \in [2^{1/16}, q]$ be a real number. If $L = \lambda^n$, then the code $\mathcal{C}$ of Construction B, with $m = \lceil \lambda \rceil$, guarantees $t(\mathcal{C}) \geqslant q/\lceil \lambda \rceil = \Omega(q/\lambda)$.

We now address the case $n \leqslant L \leqslant 2^{n/16}$. Let b be the smallest positive integer value that satisfies

$$\lfloor n/b \rfloor^b \geqslant L.$$

Construction C: Consider n q-level flash-memory cells, and a complete data graph $\mathcal{D}$ with L states. For all $1 \leqslant i \leqslant b$, let v_i be a symbol from an alphabet of size

$$\lfloor n/b \rfloor \geqslant L^{1/b}.$$

We may represent any symbol $v \in \mathcal{V}$ as a vector of symbols $(v_1, v_2, \ldots, v_b)$.

Partition the n flash-memory cells into b groups, each with $\lfloor n/b \rfloor$ cells (some cells may remain unused). Encoding the symbol v into n cells is equivalent to the encoding of each v_i into the corresponding group of $\lfloor n/b \rfloor$ cells. As the alphabet size of each v_i is at most the number of cells it is to be encoded into, we can use Construction A to store v_i. □

Notice that for $L \leqslant n$ and $b = 1$, the aforementioned construction is equivalent to Construction A.

Example 6: Consider a flash memory with $n = 16$ cells and $q = 4$ levels per cell. Let $L = 56$, and let the data graph $\mathcal{D}$ be a complete graph. To illustrate Construction C, 16 cells are partitioned into $b = 2$ groups denoted by $\mathbf{c} = (c_0, c_1, \ldots, c_7)$ and $\mathbf{c}' = (c'_0, c'_1, \ldots, c'_7)$.

Let $v \in \mathbb{Z}_L$ denote the value of the stored data. Let $v_1, v_2 \in \mathbb{Z}_8$ be defined as

$$v_1 = \lfloor v/8 \rfloor \qquad v_2 = v \bmod 8.$$

We store v_1 in $\mathbf{c}$ and v_2 in $\mathbf{c}'$ using the encoding method described in [23], which is essentially an exhaustive search for a low-weight change in the vector. If the data v change as

$$0 \to 23 \to 45 \to 6 \to 27 \to 12$$

we get the following changes in the state of the system:

v	(v_1, v_2)	$\mathbf{c}$	$\mathbf{c}'$
0	$(0,0)$	00000000	00000000
23	$(2,7)$	00100000	00000001
45	$(5,5)$	00110000	00000011
6	$(0,6)$	00111001	01000011
27	$(3,3)$	00111111	01000111
12	$(1,4)$	12111111	01111111.

□

Claim 7: For $n \leqslant L \leqslant 2^{n/16}$, it holds that

$$b \leqslant \frac{2 \log L}{\log (n/\log L)}.$$

Proof: Let $b = \frac{2 \log L}{\log (n/\log L)}$. Notice that

$$\lfloor n/b \rfloor \geqslant \frac{n \log (n/\log L)}{4 \log L}.$$

Thus

$$\begin{aligned} \log \lfloor n/b \rfloor^b &= b \log \lfloor n/b \rfloor \\ &\geqslant \frac{2 \log L}{\log (n/\log L)} \log \left(\frac{n \log (n/\log L)}{4 \log L} \right) \\ &\geqslant \frac{2 \log L}{\log (n/\log L)} \log \left(\frac{n}{4 \log L} \right) \\ &\geqslant \frac{2 \log L}{\log (n/\log L)} \log \left(\sqrt{\frac{n}{\log L}} \right) = \log L. \end{aligned}$$

We used the fact that $L \leqslant 2^{n/16}$ to establish the inequality $\frac{n}{4 \log L} \geqslant \sqrt{\frac{n}{\log L}}$ used in the last step above. ∎

Theorem 8: For $n \leqslant L \leqslant 2^{n/16}$, the number of rewrite operations the code $\mathcal{C}$ of Construction C guarantees is lower bounded by

$$t(\mathcal{C}) \geqslant \frac{n(q-1) \log (n/\log L)}{16 \log L} = \Omega \left(\frac{nq \log (n/\log L)}{\log L} \right).$$

Proof: Using Construction C, the number of possible rewrite operations is bounded by the possible number of rewrite operations for each of the b cell groups. By Theorem 4 and Claim 7, this is at least

$$\begin{aligned} \left\lfloor \frac{n}{b} \right\rfloor \cdot \frac{q-1}{8} &\geqslant \left(\frac{n \log (n/\log L)}{2 \log L} - 1 \right) \frac{q-1}{8} \\ &= \Omega \left(\frac{nq \log (n/\log L)}{\log L} \right). \end{aligned}$$

∎

C. Analysis for a Bounded Out-Degree Data Graph

We now return to the trajectory code from Section III-A when applied to the case of a data graph $\mathcal{D}$ with upper bounded out-degree δ. We refer to such a graph as δ-restricted. An example for such a graph is the De Bruijn graph of order m over an alphabet of size q in which there are q^m vertices and the out-degree of every vertex is q. We note that the out-degree of the vertices may vary from 1 to δ.

To simplify our presentation, in the theorems below, we will again use the asymptotic notation freely; however, as opposed to Section III-A, we will no longer state or make an attempt to optimize the constants involved in our calculations. We assume that $n \leqslant L$, since for $L \leqslant n$, Construction A can be used to obtain optimal codes (up to constant factors). In this section, we study the case $L \leqslant 2^{cn}$ for certain values of c. We do not address the case of larger L, as its analysis, although based on similar ideas, becomes rather tedious and overly lengthy.

Using the notation of Section III-A, to realize the trajectory code, we need to specify d as well as $n_0, n_1, \ldots, n_d$. We consider two cases: the case in which δ is *small* compared to n, and the case in which δ is *large*.

The following construction is for the case in which δ is *small* compared to n.

Construction D: The trajectory code $\mathcal{C}$ for n q-level memory cells with a δ-restricted graph $\mathcal{D}$ with vertices $\mathcal{V} = \mathbb{Z}_L$ of size L, where

$$\delta \leqslant \left\lfloor \frac{4n \log (n/\log L)}{\log L} \right\rfloor$$

is formed using $d+1$ registers, where

$$d = \lfloor \log L / 8 \log (n/\log L) \rfloor .$$

The size of the $d+1$ registers is given by

$$n_0 = \lfloor n/2 \rfloor$$

and

$$n_i = \lfloor n/(2d) \rfloor \geqslant \delta$$

for all $1 \leqslant i \leqslant d$.

The update function of Construction C is used to store an "anchor" vertex in register S_0. Each register S_i, where $1 \leqslant i \leqslant d$, is updated using Construction A to store the identities of the edges taken on the path from the anchor vertex. □

Recall that the anchor and the edges stored in $S_0, S_1, S_2, \ldots$ show how the data changes its value with rewrite operations. That is, they show the trace of the changing data in the data graph $\mathcal{D}$. Every $d+1$ rewrite operations change the data stored in the register S_i exactly once. After every $d+1$ rewrite operations, the next rewrite resets the anchor's value in S_0, and the same rewriting process starts again.

Suppose that the rewrite operations change the stored data as $v_0 \to \cdots \to v_i \to v_{i+1} \to \cdots$. With the rewriting code of Construction D, the data stored in the register S_0 change as $v_0 \to v_{d+1} \to v_{2(d+1)} \to v_{3(d+1)} \to \cdots$. For all $1 \leqslant i \leqslant d$, the data stored in the register S_i change as $(v_{i-1}, v_i) \to (v_{i-1+(d+1)}, v_{i+(d+1)}) \to (v_{i-1+2(d+1)}, v_{i+2(d+1)}) \to (v_{i-1+3(d+1)}, v_{i+3(d+1)}) \to \cdots$ Here, every edge $(v_{j-1}, v_j) \in \mathcal{E}$ is locally labeled by the alphabet $\mathbb{Z}_\delta$.

Theorem 9: Construction D guarantees the construction of a code $\mathcal{C}$ for a memory with n q-level cells with

$$t(\mathcal{C}) = \Omega(nq)$$

rewrite operations, where

$$L \leqslant 2^{\lfloor n/2 \rfloor / 16} \quad \text{and} \quad \delta \leqslant \left\lfloor \frac{4n \log (n/\log L)}{\log L} \right\rfloor .$$

Proof: It follows from Theorems 4 and 8 that the lower bound on the number of rewrite operations of register S_0 is, up to a constant factor, equal to the lower bound on S_i, where $1 \leqslant i \leqslant d$. Mathematically, we have

$$\begin{aligned} \Omega\left(\frac{n_0 q \log (n_0/\log L)}{\log L}\right) &= \Omega\left(\frac{nq \log (n/\log L)}{\log L}\right) \\ &= \Omega\left(\frac{nq}{d}\right) = \Omega\left(n_i q\right). \end{aligned}$$

Since there are $d+1$ registers that are used in a round-robin fashion, the total number of rewrite operations is lower bounded by $t(\mathcal{C}) = \Omega(nq)$. Notice that in order to use Theorems 8 and 4, we require that $n_0 \leqslant L \leqslant 2^{n_0/16}$ and $\delta \leqslant n_i$. These two assertions hold for our setting of parameters (we assume throughout this section that $L \geqslant n$). ■

Example 10: Consider a floating code, where k variables of alphabet size ℓ are stored in n q-level cells. When Construction D is used to build the floating code, we get $L = \ell^k$ and $\delta = k(\ell-1)$. So if $k(\ell-1) \leqslant \left\lfloor \frac{4n \log (n/(k \log \ell))}{k \log \ell} \right\rfloor$, the code can guarantee $t(\mathcal{C}) = \Omega(nq)$ rewrite operations, which is asymptotically optimal. □

The next construction applies to the case where $n \ll \delta$.

Construction E: Consider n q-level flash memory cells, and a δ-restricted data graph $\mathcal{D}$, with

$$\left\lfloor \frac{4n \log (n/\log L)}{\log L} \right\rfloor \leqslant \delta \leqslant L - 1.$$

For the trajectory code, let

$$d = \lfloor \log L / \log \delta \rfloor .$$

Set the size of the registers to

$$n_0 = \lfloor n/2 \rfloor$$

and

$$n_i = \lfloor n/(2d) \rfloor$$

for all $1 \leqslant i \leqslant d$.

The update and decoding functions of the trajectory code $\mathcal{C}$ are defined as follows: use the encoding scheme specified in Construction C to store an "anchor" in S_0 and store an "edge" in S_i, for all $1 \leqslant i \leqslant d$. As for the remaining details, they are the same as Construction D. □

Theorem 11: Construction E guarantees the construction of a code $\mathcal{C}$ for a memory with n q-level cells with

$$t(\mathcal{C}) = \Omega\left(\frac{nq\log(n/\log L)}{\log\delta}\right)$$

rewrite operations, where

$$L \leqslant 2^{n/64} \quad \text{and} \quad \left\lfloor \frac{4n\log(n/\log L)}{\log L} \right\rfloor \leqslant \delta \leqslant L-1.$$

Proof: It is shown in the Appendix that for the given inequalities in Theorem 11, it follows that $n_0 \leqslant L \leqslant 2^{n_0/16}$, and $n_i \leqslant \delta \leqslant 2^{n_i/16}$. Using Theorem 8, it follows that the number of rewrite operations supported by register S_0 satisfies the lower bound

$$\Omega\left(\frac{n_0 q\log(n_0/\log L)}{\log L}\right) = \Omega\left(\frac{nq\log(n/\log L)}{\log L}\right).$$

Similarly, for all $1 \leqslant i \leqslant d$, the number of rewrite operations supported in S_i is lower bounded by

$$\Omega\left(\frac{n_i q\log(n_i/\log\delta)}{\log\delta}\right) = \Omega\left(\frac{nq\log(n/\log L)}{d\log\delta}\right) = \Omega\left(\frac{nq\log(n/\log L)}{\log L}\right).$$

Thus, as in Theorem 9, we conclude that the total number of rewrite operations in the scheme outlined in Section III-A is lower bounded by $d+1$ times the bound for each register S_i, and so, $t(\mathcal{C}) = \Omega\left(\frac{nq\log(n/\log L)}{\log\delta}\right)$. ∎

D. Optimality of the Code Constructions

We now prove upper bounds on the number of rewrite operations in general rewriting schemes, which match the lower bounds induced by our code constructions. They show that our code constructions are asymptotically optimal.

Theorem 12: Any rewriting code $\mathcal{C}$ that stores symbols from some data graph $\mathcal{D}$ in n q-level flash-memory cells supports at most

$$t(\mathcal{C}) \leqslant n(q-1) = O(nq)$$

rewrite operations.

Proof: In the best case, all cells are initialized at level 0, and every rewrite increases exactly one cell by exactly one level. Thus, the total number of rewrite operations is bounded by $n(q-1) = O(nq)$. ∎

Corollary 13: The codes from Constructions A and D are asymptotically optimal.

For large values of L, we can improve the upper bound. First, let r denote the largest integer such that

$$\binom{r+n-1}{r} < L-1.$$

We use the following claim.

Claim 14: Let L satisfy $n \leqslant L \leqslant 2^{n/16}$. The following inequality holds

$$r \geqslant \frac{1}{128}\cdot\frac{\log L}{\log(n/\log L)}.$$

Proof: First, it is easy to see that $r \in [1,n]$. Now, we may use the well-known bound for all $v \geqslant u \geqslant 1$

$$\binom{v}{u} < \left(\frac{ev}{u}\right)^u$$

where e is the base of the natural logarithm. Let $m = n/r$. It follows that

$$\binom{r+n-1}{r} \leqslant \binom{r+n}{r} \leqslant \binom{2n}{r} < \frac{2^r e^r n^r}{r^r}.$$

Hence

$$\log\binom{r+n-1}{r} < r\log\left(\frac{2en}{r}\right) = \frac{n}{m}\log(2em).$$

Thus, it suffices to prove that

$$\frac{n}{m}\log(2em) \leqslant \log(L-1).$$

We conclude via basic computations that if

$$m = 128\cdot\frac{n\log(n/\log L)}{\log L}$$

then

$$\binom{r+n-1}{r} < L-1.$$

∎

Theorem 15: Let $L \leqslant 2^{n/16}$. When $L \geqslant n$, any rewriting code $\mathcal{C}$ that stores symbols from the complete data graph $\mathcal{D}$ in n q-level flash-memory cells can guarantee at most

$$t(\mathcal{C}) = O\left(\frac{nq\log(n/\log L)}{\log L}\right)$$

rewrite operations.

Proof: Let us examine a state s of the n flash-memory cells, currently storing a value $v \in \mathcal{V}$, i.e., $F_d(s) = v$. Having no constraint on the data graph, the next symbol we want to store may be any of the $L-1$ symbols $v' \in \mathcal{V}$, where $v' \neq v$.

If we allow ourselves r operations of increasing a single-cell level of the n flash-memory cells by one (perhaps operating on the same cell more than once), we may reach at most

$$\binom{n+r-1}{r}$$

distinct new states. However, by our choice of r, we have $\binom{n+r-1}{r} < L-1$. In the worst case, we need at least $r+1$ such operations to realize a rewrite. Since we have a total of

n cells with q levels each, the guaranteed number of rewrite operations is upper bounded by

$$t(\mathcal{C}) \leqslant \frac{n(q-1)}{r+1} = O\left(\frac{nq\log(n/\log L)}{\log L}\right).$$

■

Corollary 16: Construction C is asymptotically optimal.

Theorem 17: Let $2^{n/16} \leqslant L \leqslant q^n$. Any rewriting code $\mathcal{C}$ that stores symbols from the complete data graph $\mathcal{D}$ in n q-level flash-memory cells can guarantee at most

$$t(\mathcal{C}) = O(q/\lambda)$$

rewrite operations, where $\lambda = \sqrt[n]{L}$.

Proof: The proof follows from Theorem 15. In this case, we note that for $\binom{n+r-1}{r}$ to be at least of size $L = \lambda^n$, we need $r = \Omega(n\lambda)$. ■

Corollary 18: Construction B is asymptotically optimal.

Theorem 19: Let $L \leqslant 2^{n/16}$. Let $\delta \geqslant \left\lfloor \frac{4n\log(n/\log L)}{\log L} \right\rfloor$. There exists a δ-restricted data graph $\mathcal{D}$ over a vertex set of size L, such that any rewriting code $\mathcal{C}$ that stores symbols from the data graph $\mathcal{D}$ in n q-level flash-memory cells can guarantee at most

$$t(\mathcal{C}) = O\left(\frac{nq\log(n/\log L)}{\log \delta}\right)$$

rewrite operations.

Proof: We start by showing that a δ-restricted graph $\mathcal{D}$ with certain properties does not allow rewriting codes $\mathcal{C}$ that support more than $t(\mathcal{C}) = O\left(\frac{nq\log(n/\log L)}{\log \delta}\right)$ rewrite operations. We then show that such a graph indeed exists. This will conclude our proof.

Let $\mathcal{D}$ be a δ-restricted graph whose diameter d is at most $O\left(\frac{\log L}{\log \delta}\right)$. Assuming the existence of such a graph $\mathcal{D}$, consider (by contradiction) a rewriting code $\mathcal{C}$ for the δ-restricted graph $\mathcal{D}$ that allows

$$t(\mathcal{C}) = \omega\left(\frac{nq\log(n/\log L)}{\log \delta}\right)$$

rewrite operations. We use $\mathcal{C}$ to construct a rewriting code $\mathcal{C}'$ for a new data graph $\mathcal{D}' = (\mathcal{V}', \mathcal{E}')$ which has the same vertex set $\mathcal{V}' = \mathcal{V}$ but is a complete graph. The code $\mathcal{C}'$ will allow

$$t(\mathcal{C}') = \omega\left(\frac{nq\log(n/\log L)}{\log L}\right)$$

rewrite operations, a contradiction to Theorem 15. This will imply that our initial assumption regarding the possible number of rewriting operations of the code $\mathcal{C}$ is false.

The rewriting code $\mathcal{C}'$ (defined by the decoding function F'_d and the update function F'_u) is constructed by *mimicking* $\mathcal{C}$ (defined by the decoding function F_d and the update function F_u). We start by setting $F'_d = F_d$. Next, let s be some state of the flash cells. Denote $F_d(s) = F'_d(s) = v_0 \in \mathcal{V}$. Consider a rewrite operation attempting to store a new value $v_1 \in \mathcal{V}$, where $v_1 \neq v_0$. There exists a path in $\mathcal{D}$ of length d', where $d' \leqslant d$, from v_0 to v_1, which we denote by

$$v_0, u_1, u_2, \ldots, u_{d'-1}, v_1.$$

We now define

$$F'_u(s, v_1) = F_u(F_u(\ldots F_u(F_u(s, u_1), u_2)\ldots, u_{d'-1}), v_1)$$

which simply states that to encode a new value v_1, we follow the updating steps dictated by F_u on a short path from v_0 to v_1 in the data graph $\mathcal{D}$.

As $\mathcal{C}$ guarantees $t(\mathcal{C}) = \omega\left(\frac{nq\log(n/\log L)}{\log \delta}\right)$ rewrite operation, the code for $\mathcal{C}'$ guarantees at least

$$t(\mathcal{C}') = \omega\left(\frac{nq\log(n/\log L)}{d\log \delta}\right) = \omega\left(\frac{nq\log(n/\log L)}{\log L}\right)$$

rewrite operation. Here, we use the fact that $d = O\left(\frac{\log L}{\log \delta}\right)$.

What is left is to show the existence of a data graph $\mathcal{D}$ of maximum out-degree δ whose diameter d is at most $O\left(\frac{\log L}{\log \delta}\right)$. To obtain such a graph, one may simply take a rooted bi-directed tree of total degree δ and corresponding depth $O\left(\frac{\log L}{\log \delta}\right)$. ■

Corollary 20: Construction E is asymptotically optimal.

IV. Robust Rewriting Codes

In this section, we discuss the design of rewriting codes with a good expected performance in terms of the number of rewriting operations.

Let $\mathbf{v} = (v_1, v_2, v_3, \ldots, v_{n(q-1)})$ denote a sequence of rewrite values. As no more than $n(q-1)$ rewrite operations may be supported, the sequence $\mathbf{v}$ is limited to $n(q-1)$ elements.

Let $\mathcal{C}$ denote a rewriting code, which stores the data from an alphabet of size L in n q-level cells. The code $\mathcal{C}$ can only support a finite number of rewrite operations in the rewrite sequence $\mathbf{v}$. We use $t(\mathcal{C}|\mathbf{v})$ to denote the number of rewrite operations in the rewrite sequence $\mathbf{v}$ that are supported by the code $\mathcal{C}$. That is, if the code $\mathcal{C}$ can support the rewrite operations $v_1, v_2, \ldots$, up to v_k, then $t(\mathcal{C}|\mathbf{v}) = k$.

Let V denote the set of all possible rewrite sequences. If we are interested in the number of rewrite operations that a code $\mathcal{C}$ guarantees in the worst case, $t(\mathcal{C})$, then we can see that

$$t(\mathcal{C}) = \min_{\mathbf{v}\in V} t(\mathcal{C}|\mathbf{v}).$$

In this section, we are interested in the expected number of rewrite operations that a randomized code $\mathcal{C}$ can support. Let Q be some distribution over rewriting codes and let $\mathcal{C}_Q$ be a *randomized code* (namely, a random variable) with distribution Q. Let $\mathrm{E}(x)$ denote the expected value of a random variable x. We define the *expected performance* of the randomized rewriting code $\mathcal{C}_Q$ to be

$$\mathrm{E}_{\mathcal{C}_Q} = \min_{\mathbf{v}\in V} \mathrm{E}(t(\mathcal{C}_Q|\mathbf{v})).$$

Our objective is to maximize $\mathrm{E}_{\mathcal{C}_Q}$, namely, to construct a distribution Q such that for all $\mathbf{v}$, $\mathcal{C}_Q$ will allow many rewrite operations in expectation. A code $\mathcal{C}_Q$ whose $\mathrm{E}_{\mathcal{C}_Q}$ is asymptotically optimal is called a *robust code*. In this section, we will present a code $\mathcal{C}_Q$ that uses a randomized rewriting scheme, which has the property that $\mathrm{E}_{\mathcal{C}_Q} \geqslant (1-\varepsilon)n(q-1)$ for any constant $\varepsilon > 0$, which clearly is a robust code.

A. Code Construction

We first present our code construction, analyze its properties, and define some useful terms. We then show that the constructed code is robust.

For a given cell state $\mathbf{c} = (c_0, c_1, \ldots, c_{n-1}) \in \mathbb{Z}_q^n$ of a memory with n q-level cells, we define the cells' *weight* $w(\mathbf{c})$ by

$$w(\mathbf{c}) = \sum_{i=0}^{n-1} c_i.$$

Clearly, $0 \leqslant w(\mathbf{c}) \leqslant n(q-1)$. Let $F_d : \mathbb{Z}_q^n \to \mathbb{Z}_L$ be the decoding function of a rewriting code. The cell state $\mathbf{c}$ represents the data $F_d(\mathbf{c})$.

Construction F: For all $0 \leqslant i \leqslant n(q-1)-1$ and $0 \leqslant j \leqslant n-1$, let $\theta_{i,j}$ and a_i be parameters chosen from the set $\mathbb{Z}_L$. We define a rewriting code $\mathcal{C}$ by its decoding function

$$F_d(\mathbf{c}) = \left(\sum_{i=0}^{n-1} \theta_{w(\mathbf{c})-1,i} c_i + \sum_{i=0}^{w(\mathbf{c})-1} a_i \right) \bmod L.$$

For every rewrite, the rewrite operation that minimizes the cell state's weight is selected. □

By default, if $\mathbf{c} = (0, 0, \ldots, 0)$, then $F_d(\mathbf{c}) = 0$. When rewriting the data, we take a greedy approach: For every rewrite, minimize the increase of the cell state's weight. (If there is a tie between cell states of the same weight, break the tie arbitrarily.) We note that, in general, the cost of looking for the minimal change may be prohibitively high. However, as we shall later see in the construction of the robust code, a careful choice of $\theta_{i,j}$ simplifies the problem considerably.

For simplicity, we will omit the term "$\bmod L$" in all computations below that concern data values. For example, the expression for F_d in the above code construction will be simply written as

$$F_d(\mathbf{c}) = \sum_{i=0}^{n-1} \theta_{w(\mathbf{c})-1,i} c_i + \sum_{i=0}^{w(\mathbf{c})-1} a_i$$

and $F_d(\mathbf{c}) - F_d(\mathbf{c}')$ will mean $(F_d(\mathbf{c}) - F_d(\mathbf{c}')) \bmod L$.

Definition 21: Let $\mathbf{c} = (c_0, c_1, \ldots, c_{n-1}) \in \mathbb{Z}_{q-1}^n$ be a cell state. For all $0 \leqslant i \leqslant n-1$, we define $N_i(\mathbf{c})$ as

$$N_i(\mathbf{c}) = (c_0, \ldots, c_{i-1}, c_i + 1, c_{i+1}, \ldots, c_{n-1})$$

and define $e_i(\mathbf{c})$ as

$$e_i(\mathbf{c}) = F_d(N_i(\mathbf{c})) - F_d(\mathbf{c}).$$

We also define the *update vector* of $\mathbf{c}$, denoted by $u(\mathbf{c})$, as

$$u(\mathbf{c}) = (e_0(\mathbf{c}), e_1(\mathbf{c}), \ldots, e_{n-1}(\mathbf{c}))$$

and the *update diversity* of $\mathbf{c}$ as

$$\left| \{ e_0(\mathbf{c}), e_1(\mathbf{c}), \ldots, e_{n-1}(\mathbf{c}) \} \right|.$$

□

The update diversity of a cell state $\mathbf{c}$ is at most L. If it is L, it means that when the current cell state is $\mathbf{c}$, no matter what the next rewrite is, we only need to increase one cell's level by one to realize the rewrite. Specifically, if the next rewrite changes the data from $F_d(\mathbf{c})$ to v', we will change from $\mathbf{c}$ to $N_i(\mathbf{c})$ by increasing the ith cell's level by one such that

$$e_i(\mathbf{c}) = v' - F_d(\mathbf{c}).$$

For good rewriting performance, it is beneficial to make the update diversity of cell states large.

Lemma 22: Let $\mathbf{c} = (c_0, c_1, \ldots, c_{n-1}) \in \mathbb{Z}_{q-1}^n$ be a cell-state vector. Using Construction F, the update diversity of $\mathbf{c}$ is

$$\left| \left\{ \theta_{w(\mathbf{c}),i} \mid 0 \leqslant i \leqslant n-1 \right\} \right|.$$

Proof: For all $0 \leqslant i \leqslant n-1$, we have

$$\begin{aligned} e_i(\mathbf{c}) &= F_d(N_i(\mathbf{c})) - F_d(\mathbf{c}) \\ &= \sum_{j=0}^{n-1} \theta_{w(\mathbf{c}),j} c_j + \theta_{w(\mathbf{c}),i} + \sum_{j=0}^{w(\mathbf{c})} a_j \\ &\quad - \sum_{j=0}^{n-1} \theta_{w(\mathbf{c})-1,j} c_j - \sum_{j=0}^{w(\mathbf{c})-1} a_j \\ &= \theta_{w(\mathbf{c}),i} + a_{w(\mathbf{c})} + \sum_{j=0}^{n-1} (\theta_{w(\mathbf{c}),j} - \theta_{w(\mathbf{c})-1,j}) c_j. \end{aligned}$$

Only the first term $\theta_{w(\mathbf{c}),i}$ depends on i. Hence, the update diversity of $\mathbf{c}$ is

$$\left| \{ e_i(\mathbf{c}) \mid 0 \leqslant i \leqslant n-1 \} \right| = \left| \left\{ \theta_{w(\mathbf{c}),i} \mid 0 \leqslant i \leqslant n-1 \right\} \right|.$$

■

Therefore, to make the update diversity of cell states large, we can make $\theta_{w(\mathbf{c}),0}, \theta_{w(\mathbf{c}),1}, \ldots, \theta_{w(\mathbf{c}),n-1}$ take as many different values as possible. A simple solution is to let $\theta_{w(\mathbf{c}),i} = i+1$ for all i.

B. Robustness

In the following, we present a rewriting code for a memory with n q-level cells that uses a randomized rewriting scheme, where $n \geqslant L$.

We then determine the asymptotic performance of the code for $nq \geqslant L \log L$.

For all $0 \leqslant i \leqslant L-1$, we define

$$\mathcal{G}_i = \{ j \mid 0 \leqslant j \leqslant n-1, j \equiv i \pmod{L} \}.$$

For $i = 0, 1, \ldots, L-1$, we have

$$\lfloor n/L \rfloor \leqslant |\mathcal{G}_i| \leqslant \lceil n/L \rceil.$$

We define

$$h_i = \sum_{j \in \mathcal{G}_i} c_j$$

where c_j is the jth cell's level. For all i, we have

$$h_i \in \mathbb{Z}_{|\mathcal{G}_i|(q-1)}.$$

We consider $\mathcal{G}_0, \mathcal{G}_1, \ldots, \mathcal{G}_{L-1}$ as L *super cells* whose levels are $\mathbf{c}' = (h_0, h_1, \ldots, h_{L-1})$. We define the weight of the supercells as

$$w(\mathbf{c}) = \sum_{i=0}^{n-1} c_i = \sum_{i=0}^{L-1} h_i = w(\mathbf{c}').$$

Construction G: For all $0 \leqslant i \leqslant n(q-1)-1$, choose a parameter a_i independently and uniformly at random from $\mathbb{Z}_L$.

We define a randomized rewriting code $\mathcal{C}_Q$ by its decoding function

$$F_d(\mathbf{c}) = \sum_{i=0}^{L-1} i h_i + \sum_{i=0}^{w(\mathbf{c})-1} a_i. \tag{1}$$

By default, if $\mathbf{c} = (0, 0, \ldots, 0)$, then $F_d(\mathbf{c}) = 0$. When rewriting the data, we take the same greedy approach as in Construction F. □

The code of Construction G may be seen as a rewriting code that stores the data of alphabet size L in L supercells, whose decoding function is (1). Each of the supercells has either $(q-1)\lfloor n/L \rfloor + 1$ levels or $(q-1)\lceil n/L \rceil + 1$ levels.

Lemma 23: Let $\mathbf{c}' = (h_0, h_1, \ldots, h_{L-1})$ be a supercell state where $h_i \leqslant (q-1)\lfloor n/L \rfloor - 1$ for all i. Using Construction G, the update vector of the supercell state $\mathbf{c}'$ is

$$u(\mathbf{c}') = \left(a_{w(\mathbf{c}')}, 1 + a_{w(\mathbf{c}')}, \ldots, L-1+a_{w(\mathbf{c}')}\right)$$

and the update diversity of the supercell state $\mathbf{c}'$ is L.

Proof: For all i

$$N_i(\mathbf{c}') = (h_0, \ldots, h_{i-1}, h_i + 1, h_{i+1}, \ldots, h_{L-1})$$

and therefore we have

$$e_i(\mathbf{c}') = F_d(N_i(\mathbf{c}')) - F_d(\mathbf{c}') = i + a_{w(\mathbf{c}')}.$$

■

Therefore, if the current supercell state is $\mathbf{c}' = (h_0, h_1, \ldots, h_{L-1})$ where $h_i \leqslant (q-1)\lfloor n/L \rfloor - 1$, for the next rewrite, we only need to increase one supercell's level by one (which is equivalent to increasing one flash-memory cell's level by one). It is also easily seen that finding the index of the supercell to change is extremely simple. If the old value stored is $v \in \mathbb{Z}_L$ and the new value is $v' \in \mathbb{Z}_L$, then we need to increase the supercell with index i where

$$i = \left(v' - v - a_{w(\mathbf{c}')}\right) \bmod L.$$

Thus, the greedy search for the minimal change has a cost of $O(1)$ time.

Lemma 24: Let $\mathbf{c}' = (h_0, h_1, \ldots, h_{L-1})$ be a supercell state where $h_i \leqslant (q-1)\lfloor n/L \rfloor - 1$ for $0 \leqslant i < L$. Using Construction G, if $\mathbf{c}'$ is the current supercell state, then no matter which value the next rewrite changes the data to, the next rewrite will only increase one supercell's level by one, and this supercell is uniformly randomly selected from the L supercells. What is more, the selection of this supercell is independent of the past rewriting history (that is, independent of the supercells whose levels were chosen to increase for the previous rewrite operations).

Proof: Let $\mathbf{c}'$ be the current supercell state, and assume the next rewrite changes the data to v'. By Lemma 23, we will realize the rewrite by increasing the ith supercell's level by one such that $i + a_{w(\mathbf{c}')} = v' - F_d(\mathbf{c}')$. Since the parameter $a_{w(\mathbf{c}')}$ is uniformly randomly chosen from the set $\mathbb{Z}_L$, i has a uniform random distribution over $\mathbb{Z}_L$.

The same analysis holds for the previous rewrite operations. Note that $w(\mathbf{c}')$ increases with every rewrite. Since $a_0, a_1, \ldots, a_{n(q-1)-1}$ are i.i.d. random variables, the selection of the supercell for this rewrite is independent of the selection for the previous rewrite operations. ■

The aforementioned lemma holds for every rewrite sequence. We now prove that the randomized rewriting code of Construction G is *robust*.

Theorem 25: Let $\mathcal{C}_Q$ be the randomized rewriting code of Construction G. Let $\mathbf{v} = (v_1, v_2, v_3, \ldots)$ be any rewrite sequence. For any constant $\varepsilon > 0$, there exists a constant $c = c(\varepsilon) > 0$ such that if $nq \geqslant cL \log L$, then

$$\mathrm{E}(t(\mathcal{C}_Q|\mathbf{v})) \geqslant (1-\varepsilon)n(q-1)$$

and therefore $\mathcal{C}_Q$ is a robust code.

Proof: Consider L bins such that the ith bin can hold $(q-1)|\mathcal{G}_i|$ balls. We use h_i to denote the number of balls in the ith bin. Note that every bin can contain at least $(q-1)\cdot\left\lfloor \frac{n}{L} \right\rfloor$ balls and at most $(q-1)\cdot\left\lceil \frac{n}{L} \right\rceil$ balls. By Lemma 24, before any bin is full, every rewrite throws a ball uniformly at random into one of the L bins, independently of other rewrite operations. The rewriting process can always continue when no bin is full. Thus, the number of rewrite operations supported by the code $\mathcal{C}_Q$ is at least the number of balls thrown until one bin is full.

Suppose that $n(q-1) - \alpha\sqrt{nq}$ balls are thrown independently and uniformly at random into L bins, and there is no limit on the capacity of any bin. Here, we set α to be $c\sqrt{L\log L}$ for a sufficiently large constant c. For all $0 \leqslant i \leqslant L-1$, let x_i denote the number of balls thrown into the ith bin. Clearly

$$\mathrm{E}(x_i) = \frac{n(q-1)}{L} - \frac{\alpha\sqrt{nq}}{L}.$$

By the Chernoff bound

$$\Pr\left(x_i \geqslant (q-1)\cdot\left\lfloor \frac{n}{L} \right\rfloor\right) \leqslant e^{-\Omega(\alpha^2/L)}.$$

By the union bound, the probability that one or more of the L bins contain at least $(q-1)\cdot\left\lfloor \frac{n}{L} \right\rfloor$ balls is upper bounded by $Le^{-\Omega(\alpha^2/L)}$. We thus have $Le^{-\Omega(\alpha^2/L)} = 2^{-\Omega(c^2)}$.

Therefore, when $n(q-1) - \alpha\sqrt{nq}$ balls are thrown independently and uniformly at random into L bins, with high probability, all the L bins have $(q-1)\cdot\left\lfloor \frac{n}{L} \right\rfloor - 1$ or fewer balls. This suffices to conclude our assertion. Notice that our proof implies that *with high probability* (over Q), the value of $t(\mathcal{C}_Q|\mathbf{v})$ will be large. This stronger statement implies the asserted one in which we consider $E(t(\mathcal{C}_Q|\mathbf{v}))$. ■

V. Concluding Remarks

In this paper, we presented a flexible rewriting model that generalizes known rewriting models, including those used by WOM codes, floating codes, and buffer codes. We presented a novel code construction, the trajectory code, for this generalized rewriting model and proved that the code is asymptotically optimal for a wide range of parameter settings, where the performance is measured by the number of rewrite operations supported by flash-memory cells in the worst case. We also studied the expected performance of rewriting codes, and presented a randomized robust code. It will be interesting to apply these new coding techniques to wider constrained-memory applications, and combine rewriting codes with error correction. These remain as topics for future research.

Appendix

We bring here the proof of the claim made at the beginning of the proof of Theorem 11.

Lemma 26: In the setting of Theorem 11, we have $n_i \leqslant \delta$.

Proof: We first note that

$$n_i = \left\lfloor \frac{n}{2d} \right\rfloor \leqslant \frac{n}{2d} = \frac{n}{2\left\lfloor \log L / \log \delta \right\rfloor}. \tag{2}$$

We note that for all real $x \geqslant 1$, we have $\lfloor x \rfloor \geqslant x/2$. Since in our setting $\log L / \log \delta \geqslant 1$, we have (2) become

$$n_i \leqslant \frac{n \log \delta}{\log L}.$$

Thus, it suffices to show that

$$\frac{n \log \delta}{\log L} \leqslant \delta$$

or equivalently to show that

$$\frac{\delta}{\log \delta} \geqslant \frac{n}{\log L}. \tag{3}$$

In our setting, $L \leqslant 2^{n/64}$ and so

$$\log L \leqslant \frac{n}{64} \leqslant 4n \log(n/\log L)$$

and $n/\log L \geqslant 64$. Thus, by our setting

$$\delta \geqslant \left\lfloor \frac{4n \log(n/\log L)}{\log L} \right\rfloor \geqslant \frac{2n \log(n/\log L)}{\log L} \tag{4}$$

where the last inequality was achieved by using $\lfloor x \rfloor \geqslant x/2$ for $x \geqslant 1$ again. Also, we get that $\delta \geqslant 4$.

We return to proving (3). We note that for $\delta \geqslant 4$, the function $\delta / \log \delta$ is a strictly increasing function, and so, using (4), we get

$$\frac{\delta}{\log \delta} \geqslant \frac{\frac{2n \log(n/\log L)}{\log L}}{\log\left(\frac{2n \log(n/\log L)}{\log L}\right)}.$$

It now suffices to prove that the right-hand side of the equation is at least $\frac{n}{\log L}$. Simple manipulation reduces this to showing that

$$\frac{n}{\log L} \geqslant 2 \log\left(\frac{n}{\log L}\right)$$

which always holds since $2^y \geqslant 2y$ for real $y \geqslant 2$, and $\log(n/\log L) \geqslant \log 64 = 6 \geqslant 2$. ■

Lemma 27: In the setting of Theorem 11, we have $\delta \leqslant 2^{n_i/16}$.

Proof: We have $L \leqslant 2^{n/64}$ and so

$$\log \delta \leqslant \frac{n \log \delta}{64 \log L}$$

and then

$$\delta \leqslant 2^{\frac{n \log \delta}{64 \log L}} \leqslant 2^{\frac{n}{64d}}. \tag{5}$$

We note that

$$\frac{n}{2d} \geqslant \frac{n \log \delta}{2 \log L} \geqslant \frac{n}{2 \log L} \geqslant 32 \geqslant 1.$$

Remembering that $\lfloor x \rfloor \geqslant x/2$ for real $x \geqslant 1$ and combining with (5), we get

$$\delta \leqslant 2^{\frac{n}{64d}} \leqslant 2^{\left\lfloor \frac{n}{2d} \right\rfloor / 16} = 2^{n_i/16}.$$

■

References

[1] R. Ahlswede and Z. Zhang, "On multiuser write-efficient memories," *IEEE Trans. Inf. Theory*, vol. 40, no. 3, pp. 674–686, May 1994.

[2] V. Bohossian, A. Jiang, and J. Bruck, "Buffer coding for asymmetric multi-level memory," in *Proc. IEEE Int. Symp. Inf. Theory*, Nice, France, Jun. 2007, pp. 1186–1190.

[3] P. Cappelletti, C. Golla, P. Olivo, and E. Zanoni, *Flash Memories*. Norwell, MA, USA: Kluwer, 1999.

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

[5] A. Fiat and A. Shamir, "Generalized write-once memories," *IEEE Trans. Inf. Theory*, vol. IT-30, no. 3, pp. 470–480, May 1984.

[6] H. Finucane, Z. Liu, and M. Mitzenmacher, "Designing floating codes for expected performance," in *Proc. Annu. Allerton Conf. Commun, Control, Comput.*, Sep. 2008, pp. 1389–1396.

[7] F. Fu and R. W. Yeung, "On the capacity and error-correcting codes of write-efficient memories," *IEEE Trans. Inf. Theory*, vol. 46, no. 7, pp. 2299–2314, Nov. 2000.

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

[9] P. Godlewski, "WOM-codes construits à partir des codes de Hamming," *Discr. Math.*, vol. 65, no. 3, pp. 237–243, Jul. 1987.

[10] A. J. H. Vinck and A. V. Kuznetsov, "On the general defective channel with informed encoder and capacities of some constrained memories," *IEEE Trans. Inf. Theory*, vol. 40, no. 6, pp. 1866–1871, Nov. 1994.

[11] C. D. Heegard, "On the capacity of permanent memory," *IEEE Trans. Inf. Theory*, vol. IT-31, no. 1, pp. 34–42, Jan. 1985.

[12] C. D. Heegard and A. A. E. Gamal, "On the capacity of computer memory with defects," *IEEE Trans. Inf. Theory*, vol. IT-29, no. 5, pp. 731–739, Sep. 1983.

[13] A. Jiang, V. Bohossian, and J. Bruck, "Rewriting codes for joint information storage in flash memories," *IEEE Trans. Inf. Theory*, vol. 56, no. 10, pp. 5300–5313, Oct. 2010.

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

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

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

[17] A. Jiang, R. Mateescu, M. Schwartz, and J. Bruck, "Rank modulation for flash memories," *IEEE Trans. Inf. Theory*, vol. 55, no. 6, pp. 2659–2673, Jun. 2009.

[18] S. Kayser, E. Yaakobi, P. H. Siegel, A. Vardy, and J. K. Wolf, "Multiple-write WOM-codes," in *Proc. 48-th Annu. Allerton Conf. Commun., Control, Comput.*, Monticello, IL, USA, Sep. 2010.

[19] A. V. Kuznetsov and B. S. Tsybakov, "Coding for memories with defective cells," *Probl. Peredachi Inf.*, vol. 10, no. 2, pp. 52–60, Apr.–Jun. 1974.
[20] H. Mahdavifar, P. H. Siegel, A. Vardy, J. K. Wolf, and E. Yaakobi, "Nearly optimal flash codes," in *Proc. IEEE Int. Symp. Inf. Theory*, Seoul, Korea, Jun. 2009, pp. 1239–1243.
[21] F. Merkx, "WOM codes constructed with projective geometries," *Traitment du Signal*, vol. 1, no. 2, pp. 227–231, 1984.
[22] R. Micheloni, L. Crippa, and A. Marelli, *Inside NAND Flash Memories*. Berlin, Germany: Springer, 2010.
[23] R. L. Rivest and A. Shamir, "How to reuse a write-once memory," *Inf. Control*, vol. 55, pp. 1–19, Oct.–Dec. 1982.
[24] G. Simonyi, "On write-unidirectional memory codes," *IEEE Trans. Inf. Theory*, vol. 35, no. 3, pp. 663–667, May 1989.
[25] W. M. C. J. van Overveld, "The four cases of write unidirectional memory codes over arbitrary alphabets," *IEEE Trans. Inf. Theory*, vol. 37, no. 3, pp. 872–878, May 1991.
[26] F. M. J. Willems and A. J. H. Vinck, "Repeated recording for an optical disk," in *Proc. 7th Symp. Inf. Theory Benelux*, Noordwijkerhout, The Netherlands, May 1986, pp. 49–53.
[27] J. K. Wolf, A. D. Wyner, J. Ziv, and J. Korner, "Coding for a write-once memory," *AT&T Bell Labs. Tech. J.*, vol. 63, no. 6, pp. 1089–1112, Jun. 1984.
[28] Y. Wu, "Low complexity codes for writing write-once memory twice," in *Proc. IEEE Int. Symp. Inf. Theory*, Austin, TX, USA, Jun. 2010, pp. 1928–1932.
[29] Y. Wu and A. Jiang, "Position modulation code for rewriting write-once memories," *IEEE Trans. Inf. Theory*, vol. 57, no. 6, pp. 3692–3697, Jun. 2011.
[30] E. Yaakobi, S. Kayser, P. H. Siegel, A. Vardy, and J. K. Wolf, "Efficient two-write WOM-codes," in *Proc. IEEE Inf. Theory Workshop*, Dublin, Ireland, Aug. 2010, pp. 1–5.
[31] E. Yaakobi, P. H. Siegel, A. Vardy, and J. K. Wolf, "Multiple error-correcting WOM-codes," in *Proc. IEEE Int. Symp. Inf. Theory*, Austin, TX, USA, Jun. 2010, pp. 1933–1937.
[32] E. Yaakobi, P. H. Siegel, and J. K. Wolf, "Buffer codes for multi-level flash memory," presented at the IEEE Int. Symp. Inf. Theory, Toronto, ON, Canada, Jul. 2008.
[33] E. Yaakobi, A. Vardy, P. H. Siegel, and J. K. Wolf, "Multidimensional flash codes," in *Proc. Annu. Allerton Conf. Commun. Control, Comput.*, Sep. 2008, pp. 392–399.
[34] G. Zémor and G. Cohen, "Error-correcting WOM-codes," *IEEE Trans. Inf. Theory*, vol. 37, no. 3, pp. 730–734, May 1991.

Anxiao Jiang (S'00–M'05–SM'12) received the B.S. degree in electronic engineering from Tsinghua University, Beijing, China in 1999, and the M.S. and Ph.D. degrees in electrical engineering from the California Institute of Technology, Pasadena, California in 2000 and 2004, respectively.

He is currently an Associate Professor in the Computer Science and Engineering Department at Texas A&M University in College Station, Texas. His research interests include information theory, data storage, networks and algorithm design.

Prof. Jiang is a recipient of the NSF CAREER Award in 2008 for his research on information theory for flash memories and a recipient of the 2009 IEEE Communications Society Best Paper Award in Signal Processing and Coding for Data Storage.

Michael Langberg (M'07) is an Associate Professor in the Mathematics and Computer Science department at the Open University of Israel. Previously, between 2003 and 2006, he was a postdoctoral scholar in the Computer Science and Electrical Engineering departments at the California Institute of Technology. He received his B.Sc. in mathematics and computer science from Tel-Aviv University in 1996, and his M.Sc. and Ph.D. in computer science from the Weizmann Institute of Science in 1998 and 2003 respectively. His research interests include information theory, combinatorics, and algorithm design.

Moshe Schwartz (M'03–SM'10) was born in Israel in 1975. He received the B.A. (*summa cum laude*), M.Sc., and Ph.D. degrees from the Technion—Israel Institute of Technology, Haifa, Israel, in 1997, 1998, and 2004 respectively, all from the Computer Science Department.

He was a Fulbright post-doctoral researcher in the Department of Electrical and Computer Engineering, University of California San Diego, and a post-doctoral researcher in the Department of Electrical Engineering, California Institute of Technology. He now holds a position with the Department of Electrical and Computer Engineering, Ben-Gurion University of the Negev, Israel. He is currently on a sabbatical as a visiting scientist in the Research Laboratory of Electronics, MIT, in Cambridge, MA.

Dr. Schwartz received the 2009 IEEE Communications Society Best Paper Award in Signal Processing and Coding for Data Storage, and the 2010 IEEE Communications Society Best Student Paper Award in Signal Processing and Coding for Data Storage. His research interests include algebraic coding, combinatorial structures, and digital sequences.

Jehoshua Bruck (S'86–M'89–SM'93–F'01) received the B.Sc. and M.Sc. degrees in electrical engineering from the Technion-Israel Institute of Technology, Haifa, Israel, in 1982 and 1985, respectively, and the Ph.D. degree in electrical engineering from Stanford University, Stanford, CA, in 1989.

He is the Gordon and Betty Moore Professor of computation and neural systems and electrical engineering at the California Institute of Technology, Pasadena, CA. His extensive industrial experience includes working with IBM Almaden Research Center, as well as cofounding and serving as Chairman of Rainfinity, acquired by EMC in 2005; and XtremIO, acquired by EMC in 2012. His current research interests include information theory and systems and the theory of computation in biological networks.

Dr. Bruck is a recipient of the Feynman Prize for Excellence in Teaching, the Sloan Research Fellowship, the National Science Foundation Young Investigator Award, the IBM Outstanding Innovation Award, and the IBM Outstanding Technical Achievement Award. His papers were recognized in journals and conferences, including winning the 2010 IEEE Communications Society Best Student Paper Award in Signal Processing and Coding for Data Storage for his paper on codes for limited-magnitude errors in flash memories, the 2009 IEEE Communications Society Best Paper Award in Signal Processing and Coding for Data Storage for his paper on rank modulation for flash memories, the 2005 A. Schelkunoff Transactions Prize Paper Award from the IEEE Antennas and Propagation Society for his paper on signal propagation in wireless networks, and the 2003 Best Paper Award in the Design Automation Conference for his paper on cyclic combinational circuits.