Rewriting Codes for Joint Information Storage in Flash Memories

Anxiao (Andrew) Jiang, *Member, IEEE*, Vasken Bohossian, and Jehoshua Bruck, *Fellow, IEEE*

***Abstract*—Memories whose storage cells transit irreversibly between states have been common since the start of the data storage technology. In recent years, flash memories have become a very important family of such memories. A flash memory cell has q states—state $0, 1, \dots, q-1$—and can only transit from a lower state to a higher state before the expensive erasure operation takes place. We study rewriting codes that enable the data stored in a group of cells to be rewritten by only shifting the cells to higher states. Since the considered state transitions are irreversible, the number of rewrites is bounded. Our objective is to maximize the number of times the data can be rewritten. We focus on the joint storage of data in flash memories, and study two rewriting codes for two different scenarios. The first code, called *floating code*, is for the joint storage of multiple variables, where every rewrite changes one variable. The second code, called *buffer code*, is for remembering the most recent data in a data stream. Many of the codes presented here are either optimal or asymptotically optimal. We also present bounds to the performance of general codes. The results show that rewriting codes can integrate a flash memory's rewriting capabilities for different variables to a high degree.**

***Index Terms*—Coding theory, data storage, flash memory.**

I. Introduction

Memories whose storage cells transit irreversibly between states have been common since the beginning of the data storage technology. Examples include punch cards and digital optical discs, where a cell can change from a 0-state to a 1-state but not vice versa. In recent years, flash memories and some other nonvolatile EEPROM's based on floating-gate cells have become a very important family of such memories. They have good properties including high data density, fast reading speed, physical robustness, etc., and have been widely used in mobile, embedded as well as mass storage devices.

We use flash memories as a typical example to explain the basic storage mechanisms of floating-gate cells. A flash memory consists of floating-gate cells as its basic storage elements. In some flash memories, a cell has two states and is called a single-level cell (SLC); but to increase data density, multilevel cells (MLCs)—where a cell has 4 to 16 or even more states—are being actively developed. For a cell with q states, we denote its states by $0, 1, \dots, q-1$.

To write (i.e., *program*) a cell, the hot-electron injection mechanism or the Fowler-Nordheim tunneling mechanism is used to inject charge (e.g., electrons) into the cell, where the charge is trapped [4]. The amount of charge trapped in a cell determines the threshold voltage of the cell: the more trapped charge, the higher the threshold voltage. The amount of trapped charge is made to concentrate around q discrete levels, corresponding to the q cell states. The state of a cell can be read by measuring the threshold voltage. Programming and reading cells are fast; however, rewriting data is much more complex. Most of the time, it requires moving cells to lower states for rewriting data, which means to remove charge from the cells. In flash memories, cells are organized as blocks. A typical block stores 64 to 256 kilobytes of data. Due to circuit complexity reasons, to rewrite, first the whole block has to be erased (which means to lower all the cells of the block to the 0-state), then all the cells are reprogrammed. This happens even if only one cell really needs to lower its state for the rewriting, and it leads to a rewriting speed substantially slower than reading. Therefore, it will be very beneficial to design codes for storing data such that the data can be rewritten many times before the block has to be erased. Reducing the number of block erasure operations is critical not only for improving rewriting speed, but also for the flash memory's longevity. Every erasure reduces the quality of the cells, and currently, a flash memory's lifetime is bounded by about $10^4 \sim 10^5$ program-erase cycles. Although technically speaking, a cell can return to a lower state through block erasures, in this paper, we are interested in the writing and rewriting of data between two block erasure operations. In that period, the cells can only go from lower states to higher states.

We model the memory introduced above with the following Write Asymmetric Memory (WAM) model. A WAM consists of n cells, where each cell has q states: state $0, 1, \dots, q-1$. Such a cell is called a q-ary cell. A cell can change from state i to state j if and only if $i < j$.

WAM is a straightforward generalization of the Write Once Memory (WOM) model, firstly introduced by Rivest and Shamir in their seminal paper [25], where $q = 2$. It is also a special case of the Generalized WOM model [7], where the state transition diagram of a cell can be any directed acyclic graph. WAM models the NOR flash memories well, where the cells in every block can be individually programmed [4]. NOR flash memories

Manuscript received October 29, 2009; revised May 31, 2010. Date of current version September 15, 2010. This work was supported in part by the Caltech Lee Center for Advanced Networking, in part by NSF Grant ECCS-0802107, and in part by the NSF CAREER Award 0747415. The material in this paper was presented in part at the IEEE International Symposium on Information Theory (ISIT'07), Nice, France, June 24–29, 2007, and in part at ISIT'08,Toronto, ON, Canada, July 6–11, 2008.

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).

V. Bohossian was with the Department of Electrical Engineering, California Institute of Technology, Pasadena, CA 91125 USA (e-mail: vincent@paradise.caltech.edu).

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

Communicated by G. D. Cohen, Associate Editor for Coding Theory.

Digital Object Identifier 10.1109/TIT.2010.2059530

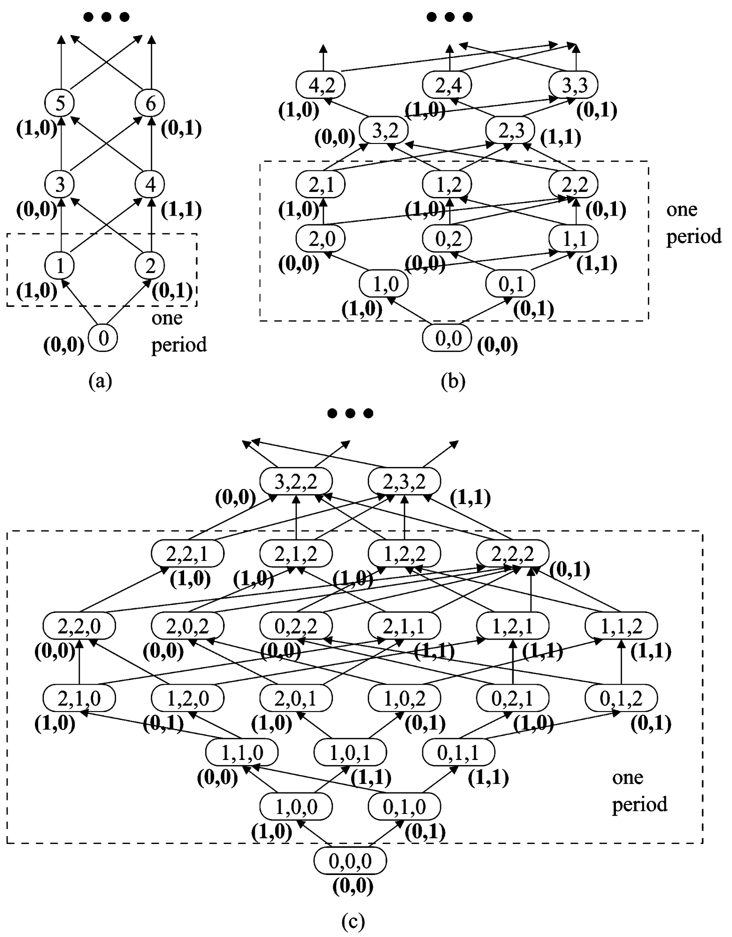

Fig. 1. Three optimal floating codes for storing two binary variables in n cells of q states. (a) $n = 1$. (b) $n = 2$. (c) $n = 3$.

are widely used for code storage, RFIDs, etc. There is a second family of popular flash memories, called NAND flash memories, which allow every page of cells (a portion of a block) to be programmed at most a few times (called partial writing of pages) between two erasures [4]. The rewriting codes for WAM have the potential to help NAND flash memories achieve a better balance between the capability to update data and the storage capacity, which can be useful in many applications.

There has been substantial research on WOM codes, where a single variable is stored in a WOM, and the code enables the variable to be rewritten numerous times. In practice, a memory stores many—let's say k—variables. A simple approach to use WOM codes in a memory is to partition it into k parts, where each part stores a variable independently. This simple approach, however, has a serious limitation. If the sequence of rewriting is very nonuniform across the variables, which is common in many applications, the WAM will be unusable soon because its longevity is determined by the most frequently rewritten variable. Therefore, it will be beneficial to integrate the rewriting capabilities of the variables, so that they can be rewritten many times regardless of what the rewriting sequence is. As we will show in this paper, such an integration is feasible, often to a high degree. We call this approach the joint storage of data in WAM.

In this paper, we study two types of rewriting codes for two different scenarios. The objective of both codes is to maximize the number of rewrites. The first code, called *floating code*, is for the joint storage of multiple variables, where every rewrite changes one variable. We show an example in Fig. 1, where two binary variables (bits) are stored in n cells of q states. The n numbers inside each circle are the states of the n cells, and the two numbers beside it are the two variables. The directed edges show how the cell states change with rewrites. For example, when $n = 3$, if the rewrites change the two binary variables as $(0,0) \to (1,0) \to (1,1) \to (0,1) \to \cdots$, the three cells' states can change as $(0,0,0) \to (1,0,0) \to (1,0,1) \to (1,0,2) \to \cdots$ The codes in Fig. 1 support $(n-1)(q-1) + \lfloor \frac{q-1}{2} \rfloor$ rewrites even in the worst case. It will be shown later that they are optimal.

The second code, called *buffer code*, is for remembering the most recent values of a changing variable. Let v be a variable with alphabet $\{0, 1, \ldots, \ell - 1\}$, whose value is changed by rewrites as $u_1 \to u_2 \to u_3 \to \cdots$ over time. Let $r \geqslant 1$ be an integer. The buffer code uses n cells of q states to remember the most recent r values of v. That is, when the variable is rewritten as u_i, the buffer code records the vector $(u_{i-r+1}, u_{i-r+2}, \ldots, u_i)$. (By convention, we let $u_j = 0$ if $j \leqslant 0$.) In other words, the buffer code remembers the most recent r values in a data stream. Buffer codes can be used to record logged data in file/database systems, to checkpoint states, or to work as a buffer for data streaming applications. We show an example in Fig. 2, where the r recent values of a binary variable are stored in an 8-ary

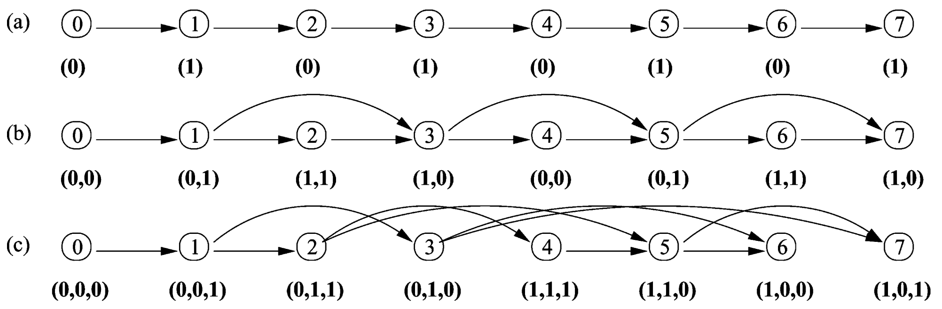

Fig. 2. Three buffer codes with parameters $n = 1, q = 8$, and $\ell = 2$. (a) $r = 1$. (b) $r = 2$. (c) $r = 3$.

cell. Suppose $r = 2$ and the binary variable changes as $1 \to 1 \to 0 \to 1 \to 0$, which means the $r = 2$ recent values change as $(0,1) \to (1,1) \to (1,0) \to (0,1) \to (1,0)$. By Fig. 2(b), we see that the cell's state can change as $1 \to 2 \to 3 \to 5 \to 7$. It will be shown that the codes in Fig. 2 can be generalized to support $\lfloor \frac{q}{2^{r-1}} \rfloor + r - 2$ rewrites when the r recent values of a binary variable are stored in a q-ary cell. Buffer codes with more cells will also be presented.

The rest of the paper is organized as follows. Section II presents an overview of the related work. Section III defines the floating code and presents bounds for its rewriting performance. Section IV presents a set of constructions for floating codes. Section V studies the buffer code. Section VI shows the conclusions.

II. Overview of Related Work

WOM was first studied by Rivest and Shamir in their original work [25], where a single variable is stored in a WOM and needs to be updated multiple times. In a WOM, the cells' states can change from 0 to 1 but not in the reverse way. This memory model was later generalized in [7] and [9], where every cell can have multiple states, and the irreversable state transition can be characterized by a directed acyclic graph. The optimization objective of WOM codes is to find the best tradeoff between the storage capacity and the rewriting capability. Several bounds on WOM code performance have been derived in [7], [9], [12], [20], [25], [28], and a number of WOM codes have been designed, including linear codes, tabular codes, and several other codes in [25], linear codes in [7], a code construction based on projective geometries in [23], a coset coding method in [5], and error-correcting WOM codes in [31].

The study on constrained memories can be further traced back to coding for defective memories, including the original work by Kuznetsov and Tsybakov [19] and papers [13], [20]. In a defective memory, a defective cell gets stuck at a state and becomes unchangable. More constrained memory models include write unidirectional memory (WUM) [24], [26], [27] and write efficient memory (WEM) [1], [2], [10]. In a WUM [26], [27], each rewrite can change the cells either from 0 to 1 or from 1 to 0, but not both. In a WEM [2], every state transition of a cell is associated with a certain cost.

The floating codes and buffer codes studied in this paper generalize the previous study on constrained memories by using the joint storage of data variables. The focus is on the fundamental tradeoff between rewriting and storage capacities when both memories and data change in constrained ways. After their introduction in [3], [14], the study on floating and buffer codes has been continued by [8], [15], [17], [22], [29], [30], etc. In [8], [17], the design of floating codes with good expected performance was studied. In [17], the rewriting model for data was further generalized using directed graphs of bounded degrees.

A commonly used technique to deal with block erasures in flash memories is wear leveling [11], which means to balance the erasures for blocks by moving data around. Compared to wear leveling, the strength of rewriting codes is that they can truly minimize the total number of erasures, not just balancing them for blocks. Rewriting codes have also been studied for other new memory technologies, such as phase-change memories [21], and for new memory storage schemes, such as rank modulation [18].

III. Definition and Bounds for Floating Codes

In this section, we define floating codes, and present bounds for their rewriting performance. We also present an optimal code for two binary variables.

A. Definition

We formally define the problem we study as follows. k variables are stored in a WAM, where each variable takes its value from an alphabet of size ℓ : $\{0, 1, \ldots, \ell - 1\}$. The WAM has n q-ary cells. The q states of a cell are also called *levels*: from level 0 to level $q - 1$. Initially, all the cells are at level 0, and all the variables have the default value 0. Each rewrite updates the value of one variable. We use $(v_1, v_2, \ldots, v_k)$—which we call the variable vector—to denote the values of the k variables, where $v_i \in \{0, 1, \ldots, \ell - 1\}$. We use $(c_1, c_2, \ldots, c_n)$—which we call the cell-state vector—to denote the levels of the n cells, where $c_i \in \{0, 1, \ldots, q-1\}$. We call $\sum_{i=1}^{n} c_i$ the *weight* of the cell-state vector. A cell-state vector $(c_1, c_2, \ldots, c_n)$ is said to be *above* another cell-state vector $(c'_1, c'_2, \ldots, c'_n)$ if $c_i \geqslant c'_i$ for all i. When the cells change their states, they can only change to a state vector above the current one.

A *floating code* has two functions: the *interpretation function* $\varphi : \{0, 1, \ldots, q - 1\}^n \to \{0, 1, \ldots, \ell - 1\}^k$, and the update function $\mu : \{0, 1, \ldots, q-1\}^n \times \{1, 2, \ldots, k\} \times \{0, 1, \ldots, \ell - 1\} \to \{0, 1, \ldots, q - 1\}^n$. The function φ maps each cell-state vector to a variable vector, which is used to decode (interpret) the stored data. The function μ shows how to rewrite: given the current cell-state vector and the information on which of the k variables is to be updated to which new value, the function μ outputs the new cell-state vector. The new cell-state vector should be above the current cell-state vector and should correspond to the new values of the variables.

A floating code *supporting t rewrites* is a code that allows the variables to be rewritten at least t times in total, regardless of what the sequence of rewrites are. In other words, we focus on the worst-case performance of the codes in this paper. Given the four parameters k, ℓ, n, q, a floating code that maximizes the value of t is called *optimal*.

B. *Optimal Floating Code for $k = 2$, $\ell = 2$ and Arbitrary n, q*

We first present a floating code for $k = \ell = 2$ and arbitrary n, q. In flash memories, to reduce interference between cells, it is often desirable for cells to have similar levels [4]. The code here achieves the minimum difference between cell levels among all optimal codes, where the difference is at most two for multilevel cells (where $q > 2$). Three examples of the code are shown in Fig. 1, corresponding to $n = 1$, 2, and 3, respectively. We comment that $n = 1, 2$ are, in fact, degenerated cases; it is only when $n = 3$ or more that the code reveals the full structure of its construction.

We define the cell-state vectors of the *ith generation* to be the cell-state vectors reachable after i rewrites. In Fig. 1, all the cell-state vectors in the same generation are placed at the same horizontal level. For example, in Fig. 1(c), the cell-state vectors in the second generation are (1,1,0),(1,0,1), and (0,1,1). The codes in Fig. 1 are all for $q \to \infty$, and they all have periodic patterns; specifically, every code is a repetition of the structure shown in the dotted box labelled by "one period." To see how, notice that the first generation in the dotted box contains two cell-state vectors corresponding to two different variable vectors, and so is true for the generation of cell-state vectors directly following the dotted box; what's more, the latter two cell-state vectors can be obtained from the former two cell state vectors by raising every cell's state by 2. (For example, in Fig. 1(b), the former two cell-state vectors are (1,0) (0,1) and; when we raise every cell's level by 2, we get (3,2) and (2,3), the latter two cell-state vectors.) The code is built for arbitrarily large q in the following way. A "period" in the code contains $2n-1$ generations. The second period directly follows—and has the same structure as—the first period, except that: (i) every cell's state is raised by 2; (ii) the pair of variable vectors (1,0) and (0,0) are switched, and the pair of variable vectors (0,1) and (1,1) are also switched. For $i = 1, 2, 3, \ldots$, the $(2i+1)$th (respectively, $(2i+2)$th) period has the same structure as the first (respectively, second) period except that every cell's level is raised by $4i$.

If q is finite, it is simple to get the corresponding code: just truncate the above code to the maximum generation, subject to the constraint that every cell's level is at most $q-1$.

We now present the formal construction of the floating code. First, let's define a few terms. A vector $(a_1, a_2, \ldots, a_m)$ is called an $i \sim (i+1)$ vector if $a_j = i$ or $i+1$ for $j = 1, 2, \ldots, m$. An $i \sim (i+1)$ vector $(a_1, a_2, \ldots, a_m)$ is called *monotonic* if its entries monotonically decreases, that is, $a_{j_1} \geqslant a_{j_2}$ for all $1 \leqslant j_1 < j_2 \leqslant m$. An $i \sim (i+1)$ vector $(a_1, a_2, \ldots, a_m)$ is called *nearly monotonic* if there exists an integer $j \in \{1, 2, \ldots, m-1\}$ such that $a_j = i$, $a_{j+1} = i+1$ and the vector $(a_1, \ldots, a_{j-1}, a_{j+1}, \ldots, a_m)$ is monotonic. For example, (1,1,0,1,0,0) is a nearly monotonic $0 \sim 1$ vector. Given any vector $A = (a_1, a_2, \ldots, a_m)$, define $\mathit{count}(A, i) = |\{j | a_j = i, 1 \leqslant j \leqslant m\}|$.

Construction 1 (Optimal Floating Code for $k = \ell = 2$ and Arbitrary n, q): For $i = 1, 2, \ldots, (n-1)(q-1) + \left\lfloor \frac{q-1}{2} \right\rfloor$, let G_i denote the set of ith generation cell-state vectors that represent the variable vector (1,0) (if i is odd) or (0,0) (if i is even), and let H_i denote the set of ith generation cell-state vectors that represent the variable vector (0,1) (if i is odd) or (1,1) (if i is even). The elements of G_i and H_i are defined as follows:

1) CASE ONE: $i \bmod (2n-1) \in \{1, 2, \ldots, n-1\}$.
In this case, A cell-state vector C is in G_i if C is a monotonic $2\left\lfloor \frac{i}{2n-1} \right\rfloor \sim \left(2\left\lfloor \frac{i}{2n-1} \right\rfloor + 1\right)$ vector and $\mathit{count}\left(C, \left(2\left\lfloor \frac{i}{2n-1} \right\rfloor + 1\right)\right) = i \bmod (2n-1)$. The cell-state vector C is in H_i if "monotonic" is replaced by "nearly monotonic" in the above condition.

2) CASE TWO: $i \bmod (2n-1) \in \{n, n+1, \ldots, 2n-3\}$.
In this case, A cell-state vector $C = (c_1, c_2, \ldots, c_n)$ is in G_i if it satisfies two conditions: (1) $\mathit{count}\left(C, 2\left\lfloor \frac{i}{2n-1} \right\rfloor\right) = 1$, $\mathit{count}\left(C, 2\left\lfloor \frac{i}{2n-1} \right\rfloor + 1\right) = 2n - (i \bmod (2n-1)) - 2$, and $\mathit{count}\left(C, 2\left\lfloor \frac{i}{2n-1} \right\rfloor\right) + 2) = (i \bmod (2n-1)) - (n-1)$; (2) if j is the unique integer such that $c_j = 2\left\lfloor \frac{i}{2n-1} \right\rfloor$, then the vector of length $n-1$
$$(c_1, \ldots, c_{j-1}, c_{j+1}, \ldots, c_n)$$
is a monotonic $\left(2\left\lfloor \frac{i}{2n-1} \right\rfloor + 1\right) \sim \left(2\left\lfloor \frac{i}{2n-1} \right\rfloor) + 2\right)$ vector.
The cell-state vector C is in H_i if "monotonic" is replaced by "nearly monotonic" in the second condition above.

3) CASE THREE: $i \bmod (2n-1) \in \{2n-2\}$.
In this case, a cell-state vector C is in G_i if one of its entries is $2\left\lfloor \frac{i}{2n-1} \right\rfloor$ and the other $n-1$ entries are $2\left\lfloor \frac{i}{2n-1} \right\rfloor + 2$. The cell-state vector C is in H_i if two of its entries are $2\left\lfloor \frac{i}{2n-1} \right\rfloor + 1$ and the other $n-2$ entries are $2\left\lfloor \frac{i}{2n-1} \right\rfloor + 2$.

4) CASE FOUR: $i \bmod (2n-1) \in \{0\}$.
In this case, a cell-state vector C is in G_i if one of its entries is $\frac{2i}{2n-1} - 1$ and the other $n-1$ entries are $\frac{2i}{2n-1}$. The cell-state vector C is in H_i if all its n entries are $\frac{2i}{2n-1}$.

It is straightforward to verify the correctness (validity) of the code in Construction 1. The key step is to verify that every ith generation cell-state vector has two $(i+1)$th cell-state vector above it that correspond to the two possible rewrite choices. For simplicity, we skip the details. The construction shows that the code has a periodic structure, with $2n-1$ rewrites as a period. To analyze its performance, we need to derive bounds for the rewriting performance of floating codes. It will be shown that the code in Construction 1 is strictly optimal.

C. *Bounds*

We show a general upper bound for t, the number of rewrites supported by floating codes, for arbitrary k, ℓ, n, and q.

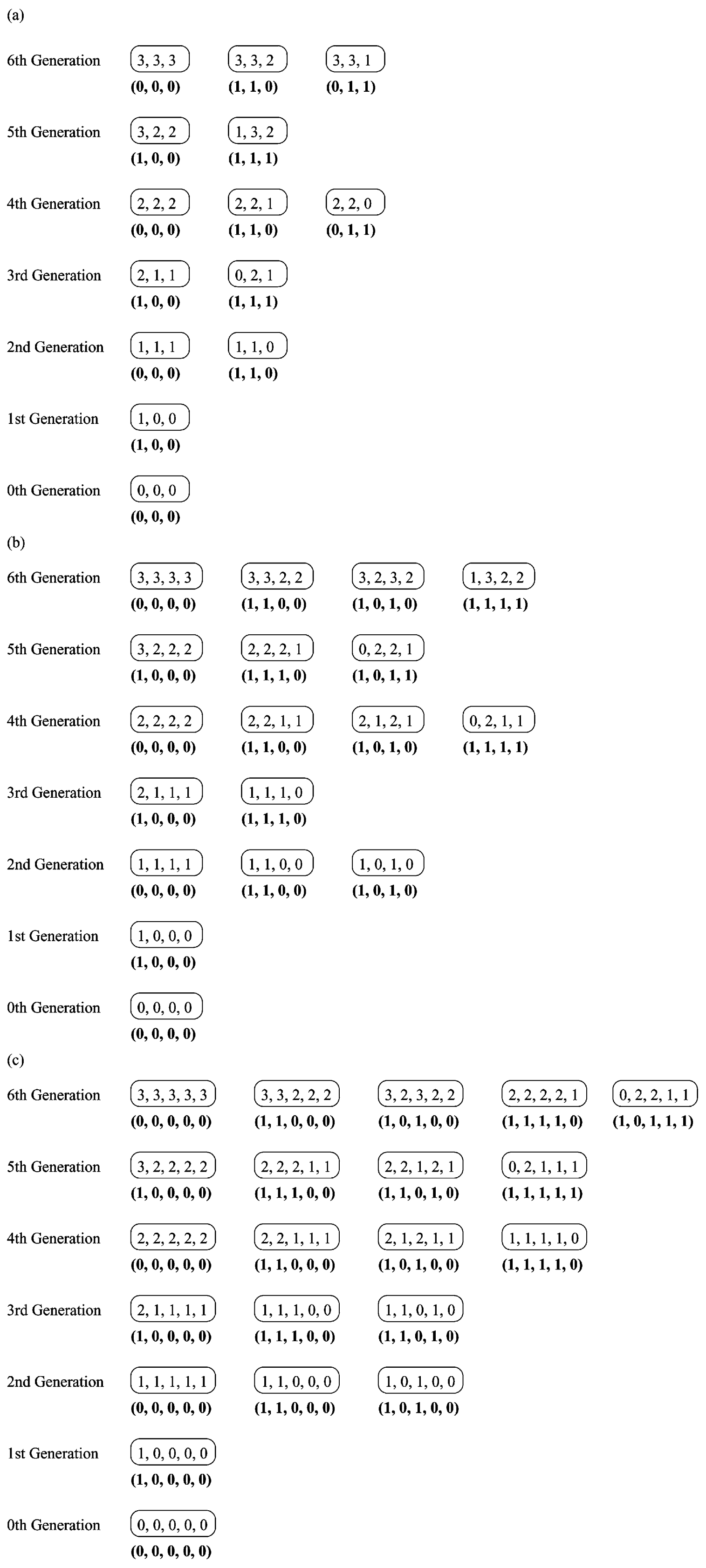

Fig. 3. A family of floating codes with $n = k \geqslant 3, \ell = 2$ and $t = 2(q-1)$. The numbers inside (respectively, beside) a node are the cell-state vector (respectively, variable vector). The code has a *cyclic property*. It takes i rewrites for the cell-state vector to reach the "ith generation." (a) $n = k = 3, q = 4$. (b) $n = k = 4$, $q = 4$. (c) $n = k = 5, q = 4$.

Theorem 2: For any *floating code*, if $n \geqslant k(\ell-1)-1$, then

$$t \leqslant [n-k(\ell-1)+1]\cdot(q-1)+\left\lfloor \frac{[k(\ell-1)-1]\cdot(q-1)}{2} \right\rfloor;$$

if $n < k(\ell-1)-1$, then

$$t \leqslant \left\lfloor \frac{n(q-1)}{2} \right\rfloor.$$

Proof: First, consider the case where $n \geqslant k(\ell-1)-1$. Let $(c_1, c_2, \ldots, c_n)$ denote the cell-state vector. Let $W_A = \sum_{i=1}^{k(\ell-1)-1} c_i$, and let $W_B = \sum_{i=k(\ell-1)}^{n} c_i$. Let's call a rewrite operation "adversarial" if it either increases W_A by at least two or increases W_B by at least one. Since there are k variables and each variable has the alphabet size ℓ, a rewrite can change the variable vector in $k(\ell-1)$ different ways. However, since W_A is the summation of only $k(\ell-1)-1$ cell levels, there are at most $k(\ell-1)-1$ ways in which a rewrite can increase W_A by one. So there must be an "adversarial" choice for every rewrite.

Consider a sequence of adversarial rewrites supported by a generic floating code. Suppose that x of those rewrites increase W_A by at least two, and y of them increase W_B by at least one. Since the maximum cell level is $q-1$, we get $x \leqslant \left\lfloor \frac{[k(l-1)-1]\cdot(q-1)}{2} \right\rfloor$ and $y \leqslant [n-k(l-1)+1]\cdot(q-1)$. So the number of rewrites supported by a floating code is at most $x+y \leqslant [n-k(l-1)+1]\cdot(q-1)+\left\lfloor \frac{[k(l-1)-1]\cdot(q-1)}{2} \right\rfloor$. The case where $n < k(\ell-1)-1$ can be analyzed similarly. ■

Corollary 3: The floating code of Construction 1 supports

$$t = (n-1)(q-1)+\left\lfloor \frac{q-1}{2} \right\rfloor$$

rewrites, which is optimal.

Proof: When $k=\ell=2$, by Theorem 2 we get $t \leqslant (n-1)(q-1)+\left\lfloor \frac{q-1}{2} \right\rfloor$ for all floating codes. It matches the value of t in Construction 1. ■

The bound in Theorem 2 is suitable for small values of k and ℓ. (For example, it is tight when $k=2$, $\ell=2$.) When k, ℓ are large, the following bound can be better.

Theorem 4: Let w be the smallest positive integer such that $\binom{w+n}{n} \geqslant \ell^k$. Then for floating codes, $t \leqslant \left\lceil \frac{(q-1)n}{w} \right\rceil k$.

Let w' be the smallest positive integer such that $\binom{w'+n}{n} > \ell^k$. Then for floating codes, when $k \geqslant 2$, $t \leqslant \left\lceil \frac{(q-1)n}{w'} \right\rceil k$.

Proof: First, consider the general case $k \geqslant 1$. Define S as $S = \{(a_1, a_2, \ldots, a_n) | \sum_{i=1}^{n} a_i \leqslant w, a_1, a_2, \ldots, a_n \text{ are non-negative integers}\}$, and let w be the smallest integer such that $|S| \geqslant \ell^k$. Define S' as $S' = \{(d_1, d_2, \ldots, d_n) | \sum_{i=1}^{n} d_i \leqslant w+n, d_1, d_2, \ldots, d_n \text{ are positive integers}\}$. By letting $d_i = a_i + 1$ for $i = 1, 2, \ldots, n$, we see that there is a one-to-one mapping between S and S'. So $|S| = |S'|$. An element $(d_1, d_2, \ldots, d_n)$ belongs to S' if and only if it is a solution to the following problem: partition a path of $w+n$ vertices into either n or $n+1$ subpaths such that for $i = 1, 2, \ldots, n$, the ith subpath has $d_i > 0$ vertices. Therefore, $|S'| = \binom{w+n}{n}$. So w is also the smallest positive integer such that $\binom{w+n}{n} \geqslant \ell^k$.

k consecutive rewrites can make the variables change to or go through any of the ℓ^k possible values. If we see a_i (for $i = 1, 2, \ldots, n$) as the increase in c_i—the state of the ith cell—and consider the way S and w are defined, we see that whatever the current cell state vector is, there exist k consecutive rewrites that increase the weight of the cell-state vector $\sum_{i=1}^{n} c_i$ by at least w. Now consider the first batch of such k rewrites, the second batch, and so on. Since the maximum weight of the cell state vector is $(q-1)n$, we get $t \leqslant \left\lceil \frac{(q-1)n}{w} \right\rceil k$.

For the slightly more restrictive case $k \geqslant 2$, we refine the above proof a little. When $k \geqslant 2$, among the cell state vectors that k consecutive rewrites can make the cell state vector change to or go through, there are at least two cell state vectors (including the current cell state vector) that correspond to the current variable vector. The rest of the proof is similar. ■

Theorem 5: There exist floating codes with $t \geqslant \left\lfloor \frac{\lfloor n/k \rfloor (q-1)}{\ell-1} \right\rfloor$.

Proof: Assign $\lfloor n/k \rfloor$ cells to each variable, where the summation of cell levels modulo ℓ represents the variable. ■

Theorem 6: When k, ℓ, q are fixed and $n \to \infty$, there exist floating codes where $t = (q-1)n + o(n)$.

Proof: See the k variables of alphabet size ℓ as one variable of alphabet size ℓ^k, and use a WOM code for rewriting. A WOM code can support $n + o(n)$ rewrites when $q = 2$ [25]. By using the q levels level by level (first use levels 0 and 1, then levels 1 and 2, and so on), $(q-1)n + o(n)$ rewrites are supported. ■

The above result shows floating codes can integrate the rewriting capability for different variables well when n is large. The next upper bound refines Theorem 4, although it requires an iterative algorithm to compute the bound.

Theorem 7: For $i = 1, 2, \ldots, k$, define s_i as follows: (1) If $\ell = 2$ and i is even, $s_i = \sum_{j=0,2,\ldots,i} \binom{k}{j}$; (2) if $\ell = 2$ and i is odd, $s_i = \sum_{j=1,3,\ldots,i} \binom{k}{j}$; (3) if $\ell > 2$, $s_i = \sum_{j=0}^{i} \binom{k}{j} (\ell-1)^j$. Also, define w_i as the smallest positive integer such that $\binom{n+w_i}{n} - \binom{n+i-1}{n} \geqslant s_i$. Let $m \in \{1, 2, \ldots, k\}$ be the integer such that $\frac{w_m}{m} \geqslant \frac{w_j}{j}$ for $j = 1, 2, \ldots, k$,.

For any *floating code*, $t \leqslant \left\lfloor \frac{n(q-1)}{w_m} \right\rfloor \cdot m + \min\{m-1, n(q-1) \bmod w_m\}$.

Proof: s_i is the number of values that the variable vector can possibly take after i rewrites. (By symmetry, s_i does not depend on the initial value of the variable vector.) Consider m consecutive rewrites. They increase the weight of the cell-state vector by at least m. For any $x \geqslant m$, the number of ways to raise the states of n cells such that the weight of the cell-state vector increases by at least m and at most x is $\binom{n+x}{n} - \binom{n+m-1}{n}$. The m consecutive rewrites can change the variable into s_m possible values, each of which corresponds to at least one way of raising the cell states. So by the definition of w_i, there is a sequence of m consecutive rewrites that increases the weight of the cell-state vector by at least w_m. Choose $\left\lfloor \frac{n(q-1)}{w_m} \right\rfloor$ such sequences of rewrites (one after another), and they make the weight of the cell state vector be at least $n(q-1) - [n(q-1) \bmod w_m]$.

After that, since the weight cannot exceed $n(q-1)$, there is a sequence of m rewrites that is not feasible due to the lack of room for the weight increase. Also, each rewrite increases the weight by at least one. So after the initial $\left\lfloor \frac{n(q-1)}{w_m} \right\rfloor$ sequences of rewrites (which consist of $\left\lfloor \frac{n(q-1)}{w_m} \right\rfloor \cdot m$ rewrites in total), at most $\min\{m-1, n(q-1) \bmod w_m\}$ more rewrites are guaranteed to be feasible. So $t \leqslant \left\lfloor \frac{n(q-1)}{w_m} \right\rfloor \cdot m + \min\{m-1, n(q-1) \bmod w_m\}$. ■

The above bound compares favorably with the upper bounds in Theorems 2 and 4 when k or ℓ is relatively large. For example, when $n=4, q=8, k=4, \ell=4$, Theorem 7 gives $t \leqslant 11$, and the bounds in Theorems 2 and 4 show $t \leqslant 14$.

IV. More Floating Code Constructions

In this section, we present several floating code constructions. They range from specific parameters to general settings.

A. Floating Code for $n = k \geqslant 3$, $\ell = 2$ and Arbitrary q

We first present a floating code for $n = k \geqslant 3$, $\ell = 2$ and arbitrary q. The code is strictly optimal when $n = k = 3$. To illustrate its structure, we show some examples of the code.

Example 8 (Floating Code for $n = k \geqslant 3$, $\ell = 2$ and Arbitrary q): Three codes are shown in Fig. 3, which support $t = 2(q-1)$ rewrites. The codes have a *cyclic property:* If the cell-state vector $(c_1 = a_1, c_2 = a_2, \ldots, c_n = a_n)$ represents the variable vector $(v_1 = b_1, v_2 = b_2, \ldots, v_k = b_k)$, then the cell-state vector $(c_1 = a_2, c_2 = a_3, \ldots, c_{n-1} = a_n, c_n = a_1)$ represents the variable vector $(v_1 = b_2, v_2 = b_3, \ldots, v_{k-1} = b_k, v_k = b_1)$. For simplicity, for every set of cell-state vectors that are cyclic shifts of each other, only one of them is shown in Fig. 3 as their representative.

For instance, consider the code in Fig. 3(c) with $n = k = 5$, $q = 4$ and $\ell = 2$. If the rewrites change the variables as $(0,0,0,0,0) \rightarrow (1,0,0,0,0) \rightarrow (1,0,1,0,0) \rightarrow (1,0,0,0,0) \rightarrow (1,0,0,0,1) \rightarrow (1,0,1,0,1) \rightarrow (1,0,1,1,1)$, the cell-state vector can change as $(0,0,0,0,0) \rightarrow (1,0,0,0,0) \rightarrow (1,0,1,0,0) \rightarrow (2,1,1,1,1) \rightarrow (2,1,1,1,2) \rightarrow (2,1,2,1,2) \rightarrow (2,1,2,2,2)$. □

We now present the general code construction. (Note that the code has a *cyclic property,* as explained in Example 8.) In the following, define $s_{\min}$ and $s_{\max}$ as the minimum and the maximum cell level, respectively. That is, $s_{\min} = \min\{c_1, c_2, \ldots, c_n\}$ and $s_{\max} = \max\{c_1, c_2, \ldots, c_n\}$.

Construction 9 (Floating Code for $n = k \geqslant 3$, $\ell = 2$ and Arbitrary q): The cell-state vectors $(c_1, c_2, \ldots, c_n)$ are mapped to the variable vectors $(v_1, v_2, \ldots, v_k)$ in the following way:

- TYPE I: If $c_1 = c_2 = \cdots = c_n$, then $v_i = 0$ for $1 \leqslant i \leqslant k$.
- TYPE II: If $s_{\max} = s_{\min} + 1$, then $v_i = c_i - s_{\min}$ for $1 \leqslant i \leqslant k$.
- TYPE III: If $(c_1, c_2, \ldots, c_n) = (s_{\min}, s_{\min} + 2, s_{\min} + 1, s_{\min} + 1, \ldots, s_{\min} + 1)$—that is, it starts with $s_{\min}$, $s_{\min} + 2$ and its next $n-2$ entries are all $s_{\min} + 1$—then $v_i = 1$ for $1 \leqslant i \leqslant k$.
- TYPE IV: If $(c_1, c_2, \ldots, c_n) = (s_{\min}, s_{\min} + 2, s_{\min} + 2, s_{\min} + 1, s_{\min} + 1, \ldots, s_{\min} + 1)$—that is, it starts with $s_{\min}$, $s_{\min} + 2$, $s_{\min} + 2$ and its next $n-3$ entries are all $s_{\min} + 1$—then $v_2 = 0$ and $v_i = 1$ for $i \neq 2$, $1 \leqslant i \leqslant k$.
- CYCLIC PROPERTY: If we cyclically shift any cell-state vector mentioned above by i positions (here $0 \leqslant i \leqslant n-1$), then the corresponding variable vector also cyclically shifts by i positions. □

For notational convenience, if C is a cell-state vector of type I (respectively, II, III, or IV), then we say that a cyclic shift of C is of type I (respectively, II, III, or IV), too. Recall that it takes i rewrites for the cell-state vector to reach the ith generation. For the code of Construction 9, the generation that a cell-state vector $C = (c_1, c_2, \ldots, c_n)$ belongs to can be computed in the following way:

- If C is of type I, it is of the $2c_1$th generation.
- If C is of type II, then let x denote the number of cells of level $s_{\max}$. C is of the $(2s_{\min} + x)$th generation.
- If C is of type III, it is of the $(2s_{\min} + n)$th generation.
- if C is of type IV, it is of the $(2s_{\min} + n + 1)$th generation.

We briefly explain how the code is used for rewriting. For a cell-state vector of the ith generation, a rewrite always changes it to the $(i+1)$th generation. For example, if $n = k = 5$ and the current cell state vector is (0,2,1,1,1) (which is of type III, fifth generation, representing $(v_1, v_2, \ldots, v_5) = (1,1,1,1,1)$), and we want to change the variable vector to (1,1,1,0,1), we can change the cell-state vector to (2,2,2,1,2) (type II, sixth generation). It is simple to verify through case enumeration that for any cell-state vector of generation $i < 2(q-1)$, there are k cell-state vectors in the $(i+1)$th generation above it that correspond to the k possible rewrite requests. So we get the following.

Theorem 10: The floating code in Construction 9 supports $t = 2(q-1)$ rewrites.

Corollary 11: The floating code in Construction 9 is optimal when $n = k = 3$.

Proof: By Theorem 2, when $n = k = 3$ and $\ell = 2$, $t \leqslant 2(q-1)$. That matches the performance of this code. ■

B. Floating Codes for $3 \leqslant k \leqslant 6$, $\ell = 2$ and Arbitrary n, q

We now use the code of Construction 1 as a building block to design floating codes for $3 \leqslant k \leqslant 6$, $\ell = 2$ and arbitrary n, q. The new codes are asymptotically optimal in n and q. The idea is to use composition of codes and a "level-by-level" approach. Recall that Construction 1 is for two binary variables. When $q = 2$, it is reduced to the following construction:

- If a cell-state vector $C = (c_1, c_2, \ldots, c_n)$ is monotonic, then it represents the variable vector (1,0) (if $\sum_{i=1}^{n} c_i$ is odd) or (0,0) (if $\sum_{i=1}^{n} c_i$ is even). If a cell-state vector $C = (c_1, c_2, \ldots, c_n)$ is nearly monotonic, then it represents the variable vector (0,1) (if $\sum_{i=1}^{n} c_i$ is odd) or (1,1) (if $\sum_{i=1}^{n} c_i$ is even).

The above code supports $n-1$ rewrites, and each rewrite changes one cell level from 0 to 1. Furthermore, it essentially uses the cells from left to right: after i rewrites, the i cells of level 1 must be among the first $i+1$ cells. This property shows that we can use n cells to encode $k = 4$ binary variables this way: "use the cells from left to right to encode two variables, and use the cells from right to left to encode the other two variables; stop rewriting when there are only three cells of level 0 left." (Note that when there are three or more cells of level 0, we can differentiate the encoding on the two ends.) For example, when $n = 9$, the cell-state vector (1,1,1,0,0,0,1,1,0) represents the variable vector (1,0,1,1), because in Construction 1, the cell-state vector $(1, 1, 1, 0, \ldots)$ represents variables (1,0) and the cell-state vector $(0, 1, 1, 0, \ldots)$ represents the variables (1,1). This approach can encode 4 binary variables and support $n - 3$ rewrites.

When $q > 2$, we can use a "level-by-level" approach: first use level 0 and level 1 to encode variables, then use level 1 and level 2, then level 2 and level 3 $\cdots$ Each rewrite raises only one cell's level except during the transition from levels i and $i + 1$ to levels $i + 1$ and $i + 2$.

Example 12: Let $k = 4$, $\ell = 2$, $n = 7$, $q = 4$. If the variable vector changes as $(0, 0, 0, 0) \rightarrow (1, 0, 0, 0) \rightarrow (1, 1, 0, 0) \rightarrow (1, 1, 1, 0) \rightarrow (0, 1, 1, 0) \rightarrow (0, 1, 0, 0) \rightarrow (0, 1, 0, 1) \rightarrow \cdots$, the cell states change as $(0, 0, 0, 0, 0, 0, 0) \rightarrow (1, 0, 0, 0, 0, 0, 0) \rightarrow (1, 0, 1, 0, 0, 0, 0) \rightarrow (1, 0, 1, 0, 0, 0, 1) \rightarrow (1, 0, 1, 1, 0, 0, 1) \rightarrow (1, 2, 1, 1, 1, 1, 1) \rightarrow (1, 2, 1, 1, 1, 2, 1) \rightarrow \cdots$ □

Theorem 13: For the above floating code with $k = 4$ and $\ell = 2$, if n is even, it supports $t = (n - 6)(q - 1) + 3$ rewrites; if n is odd, it supports $t = (n - 5)(q - 1) + 2$ rewrites.

Proof: Every rewrite raises one cell's level by one, unless the rewrite causes the transition from levels i and $i + 1$ to levels $i + 1$ and $i + 2$. During that transition, if n is even (respectively, odd), at most four (respectively, three) cells need to be set to level $i + 2$. So the first pair of cell levels support $n - 3$ rewrites and every subsequent pair of cell levels support at least $n - 6$ (if n is even) or $n - 5$ (if n is odd) rewrites. ■

When there are $k = 3$ binary variables, the coding becomes easier. That is because we can use a monotonic vector to encode a single binary variable, and with every rewrite of the variable we can increase one cell level by one. So we can partition the three variables into two and one, and use the same composition method in coding. When there are $k = 5$ or 6 binary variables, we can use the idea proposed by Fiat and Shamir in [7]: partition the n cells into three parts—the two "ends" and the "middle part"—and use each part to encode some data; when one "end" nearly meets the "middle part", allocate a new "middle part." Here, we can use each "end" to encode two binary variables, and use the "middle part" to encode the remaining one or two binary variables. For succinctness, we skip the detailed code construction. Readers can refer to [7] for the original idea (for WOM codes) and [16] for the detailed floating-code construction and analysis. The codes lead to the following theorem.

Theorem 14: When $k = 3$, $\ell = 2$ and $n \geqslant 5$, if n is even, there is a floating code with $t = (n - 4)(q - 1) + 2$; if n is odd, there is a floating code with $t = (n - 3)(q - 1) + 1$. When $k = 5$, $\ell = 2$ and $n \geqslant 9$, there is a floating code with $t > (n - 10 - 2\log_2 n)(q - 1) + 3$. When $k = 6$, $\ell = 2$ and $n \geqslant 12$, there is a floating code with $t > (n - 17 - 6\log_2 n)(q - 1) + 5$.

All the codes presented here support $t = n(q - 1) - o(nq)$ rewrites. Since every rewrite raises cell levels (up to $q - 1$), the codes are all asymptotically optimal in n, the number of cells, and in q, the number of cell levels.

C. Floating Codes for $\ell = 2$ and General k, n, q

The codes shown so far are either for a small number of binary variables (i.e., $k \leqslant 6$) or for the special case $k = n$. In this subsection, we present a floating code for general values of k. The code is asymptotically optimal in n and q.

Construction 15 (Indexed Code: Floating Code for $\ell = 2$ and General k, n, q): Divide the k variables into a groups: $g_1, g_2, \ldots, g_a$. For the n cells, set aside a small number of cells as index cells and divide the other cells into b groups: $h_1, h_2, \ldots, h_b$. Here a and b are chosen parameters, and $b \geqslant a$. For $1 \leqslant i \leqslant a$, the variables of g_i are coded using a floating code and are stored in h_i. Afterwards, every time a cell group can no longer support any more rewriting (say it stores g_i), store g_i in the next unused cell group. The index cells are used to remember which cell group stores which variable group. □

Theorem 16: When $k \log_q k = o(n)$ and $k = \Omega(q)$, the indexed code can support

$$n(q - 1)\left(1 - \frac{1}{\Omega(\sqrt{n/(k\log_q k)})}\right) = n(q - 1)(1 - o(1))$$

rewrites, which is asymptotically optimal.

Proof: Let α denote $\sqrt{\frac{n}{k\log_q k}}$. In the indexed code construction, let $a = k/2$, $b = \alpha k$, and set $b\log_q k$ as the number of index cells. Let each variable group g_i have two binary variables, and let each cell group h_i have $\alpha\log_q k - \log_q \frac{k}{2}$ cells. Clearly, the number of index cells is sufficient, since every time a new cell group h_i is used, we can use $\log_q \frac{k}{2}$ index cells to show which variable group it encodes. By Construction 1, every cell group h_i can support $\left(\alpha\log_q k - \log_q \frac{k}{2} - 1\right)(q - 1) + \left\lfloor\frac{q-1}{2}\right\rfloor$ rewrites. When the rewriting process ends, at most $\frac{k}{2} - 1$ cell groups will be underutilized. So the code can support at least $\left[b - \left(\frac{k}{2} - 1\right)\right]\left[\left(\alpha\log_q k - \log_q \frac{k}{2} - 1\right)(q - 1) + \left\lfloor\frac{q-1}{2}\right\rfloor\right] = \left(\alpha k - \frac{k}{2} + 1\right)\left[\left(\alpha\log_q k - \log_q \frac{k}{2} - 1\right)(q - 1) + \left\lfloor\frac{q-1}{2}\right\rfloor\right] = n(q - 1)\left(1 - \frac{1}{\alpha} + o\left(\frac{1}{\alpha}\right)\right) = n(q - 1)\left(1 - \frac{1}{\sqrt{n/k\log_q k}} + o\left(\frac{1}{\sqrt{n/k\log_q k}}\right)\right) = n(q - 1)(1 - o(1))$ rewrites. ■

The above proof shows a simple way to use index cells to record the mapping between cell groups and variable groups. In practice, the code can be further refined using the fact that at any moment, there are only a partially used cell groups, so the mapping is a permutation. For simplicity, we skip the details. Interested readers can refer to the discussion in [15].

D. Constructions Based on Covering Codes for General k, ℓ, n, q

We have so far focused on floating codes for binary variables. In this subsection, we present a new method that converts floating codes with large alphabets to floating codes with small alphabets (including the binary alphabet) by using covering codes. The idea is to map a variable with a large alphabet to a vector of a small alphabet such that when the variable changes its value (i.e., is rewritten), only a few (preferably one) entries in the vector change their values. Based on this method, we can obtain a series of bounds and code constructions for large alphabets.

Construction 17 (Mapping Based on Linear Covering Codes): Let v be a variable of alphabet size ℓ. Choose an (n_0, k_0) linear covering code of alphabet size ℓ_0, which has length n_0 and dimension k_0. The requirement is $\ell_0^{n_0-k_0} \geqslant \ell$. The code has $\ell_0^{n_0-k_0}$ cosets of the codewords. Among them, choose any ℓ cosets, and map them to the ℓ values of v. □

Example 18: Let v be a variable that takes its value from an alphabet of size $\ell = 4$: $\{0, 1, 2, 3\}$. Choose the simple (3,1) repetition code. As a result, the mapping from v to bit vectors of length 3 is as follows:

vector	$(0,0,0)$ $(1,1,1)$	$(1,0,0)$ $(0,1,1)$	$(0,1,0)$ $(1,0,1)$	$(0,0,1)$ $(1,1,0)$
v	0	1	2	3

To design a floating code for variables $v_1, v_2, \ldots, v_k$ of alphabet size 4, we first map them to binary variables $\{w_{i,j} | 1 \leqslant i \leqslant k, 1 \leqslant j \leqslant 3\}$, where each binary vector $(w_{i,1}, w_{i,2}, w_{i,3})$ represents v_i. Then we use a floating code for the $3k$ binary variables. Every rewrite for $(v_1, v_2, \ldots, v_k)$ maps to exactly one rewrite for $(w_{1,1}, w_{1,2}, \ldots, w_{k,3})$. (For instance, if $k = 2$ and (v_1, v_2) changes as $(0,0) \rightarrow (0,3) \rightarrow (0,2) \rightarrow (3,2) \rightarrow (3,1)$, the binary vector $(w_{1,1}, w_{1,2}, w_{1,3}, w_{2,1}, w_{2,2}, w_{2,3})$ will correspondingly change as $(0,0,0,0,0,0) \rightarrow (0,0,0,0,0,1) \rightarrow (0,0,0,1,0,1) \rightarrow (0,0,1,1,0,1) \rightarrow (0,0,1,1,0,0)$.) So if the floating code supports t rewrites for the binary variables, it also supports t rewrites for the 4-ary variables $v_1, v_2, \ldots, v_k$. □

It is important for the selected covering code to have a small covering radius, because when the large-alphabet variable changes, the covering radius of the code equals the number of entries in the small-alphabet vector that may change.

Let R denote the covering radius of the (n_0, k_0) covering code in Construction 17. Let $t(n, q, k, \ell)$ denote the greatest number of rewrites that a floating code can guarantee to support, when k ℓ-ary variables are stored in n cells with q states. (Namely, $t(n, q, k, \ell)$ is the optimal value of t for floating codes with parameters n, q, k, ℓ.) The following theorem compares the coding performance for different alphabets.

Theorem 19: $t(n, q, k, \ell) \geqslant \lfloor t(n, q, kn_0, \ell_0)/R \rfloor$.

Proof: Map the variables $v_1, v_2, \ldots, v_k$ of alphabet size ℓ to kn_0 variables of alphabet size ℓ_0 with Construction 17. Build an optimal floating code C for the kn_0 variables of alphabet size ℓ_0, which guarantees $t(n, q, kn_0, \ell_0)$ rewrites.

For the (n_0, k_0) covering code, every vector of length n_0 is within Hamming distance R from a codeword. So by the symmetry of linear codes, for every vector and each of the $\ell_0^{n_0-k_0}$ cosets, there is a vector in the coset that is within Hamming distance R from the former vector. So when we rewrite $v_i (1 \leqslant i \leqslant k)$, we are correspondingly rewriting at most R ℓ_0-ary variables. So C supports $\lfloor t(n, q, kn_0, \ell_0)/R \rfloor$ rewrites for $v_1, v_2, \ldots, v_k$. So $t(n, q, k, \ell) \geqslant \lfloor t(n, q, kn_0, \ell_0)/R \rfloor$. ■

1. For $m \geqslant 2$, $\ell \leqslant 2^m$, $t(n,q,k,\ell) \geqslant t(n,q,k(2^m-1),2)$.
2. For $\ell \leqslant 2^{11}$, $t(n,q,k,\ell) \geqslant \lfloor t(n,q,23k,2)/3 \rfloor$.
3. For $a \geqslant b \geqslant 1$ and $\ell \leqslant 2^{a-b}$, $t(n,q,k,\ell) \geqslant \lfloor t(n,q,ka,2)/\lceil \frac{a-b}{2} \rceil \rfloor$.
4. For $b \geqslant 4$, $a \geqslant 2^{b-2}$ and $\ell \leqslant 2^{a-b}$, $t(n,q,k,\ell) \geqslant \lfloor t(n,q,ka,2)/(\lfloor \frac{a}{2} \rfloor - 2^{(b-4)/2}) \rfloor$.
5. For $\ell \leqslant 2^7$, $t(n,q,k,\ell) \geqslant \lfloor t(n,q,23k,2)/2 \rfloor$.
6. For $\ell \leqslant 2^{19}$, $t(n,q,k,\ell) \geqslant \lfloor t(n,q,47k,2)/5 \rfloor$.
7. For all $a \geqslant 1$ and $\ell \leqslant 2^{a-1}$, $t(n,q,k,\ell) \geqslant \lfloor t(n,q,ka,2)/\lfloor \frac{a}{2} \rfloor \rfloor$.
8. For all $a \geqslant 2$ and $\ell \leqslant 2^{a-2}$, $t(n,q,k,\ell) \geqslant \lfloor t(n,q,ka,2)/\lfloor \frac{a-1}{2} \rfloor \rfloor$.
9. For all $a \geqslant 3$ and $\ell \leqslant 2^{a-3}$, $t(n,q,k,\ell) \geqslant \lfloor t(n,q,ka,2)/\lfloor \frac{a-2}{2} \rfloor \rfloor$.
10. For $a \geqslant 6$, $\ell \leqslant 2^{a-4}$, $t(n,q,k,\ell) \geqslant \lfloor t(n,q,ka,2)/\lfloor \frac{a-4}{2} \rfloor \rfloor$.
11. For $a \geqslant 7$, $\ell \leqslant 2^{a-5}$, $t(n,q,k,\ell) \geqslant \lfloor t(n,q,ka,2)/\lfloor \frac{a-5}{2} \rfloor \rfloor$.
12. For $a \geqslant 14$, $\ell \leqslant 2^{a-6}$, $t(n,q,k,\ell) \geqslant \lfloor t(n,q,ka,2)/\lfloor \frac{a-8}{2} \rfloor \rfloor$
13. For $a \geqslant 19$, $\ell \leqslant 2^{a-7}$, $t(n,q,k,\ell) \geqslant \lfloor t(n,q,ka,2)/\lfloor \frac{a-9}{2} \rfloor \rfloor$.
14. For all $b \geqslant 2$, $a \geqslant 2^{2b}-1$ and $\ell \leqslant 2^{a-2b-1}$, $t(n,q,k,\ell) \geqslant \lfloor t(n,q,ka,2)/\lfloor \frac{a-2^b}{2} \rfloor \rfloor$.
15. For all $b \geqslant 2$, a even and $\ell \leqslant 2^{a-2b}$, $t(n,q,k,\ell) \geqslant \lfloor t(n,q,ka,2)/\lfloor \frac{a-2^{(2b-1)/2}}{2} \rfloor \rfloor$.
16. For all $b \geqslant 2$, a odd and $\ell \leqslant 2^{a-2b}$, $t(n,q,k,\ell) \geqslant \lfloor t(n,q,ka,2)/\lfloor \frac{a-2^{(2b-1)/2}-1}{2} \rfloor \rfloor$.
17. For $a \geqslant 127$ and $\ell \leqslant 2^{a-8}$, $t(n,q,k,\ell) \geqslant \lfloor t(n,q,ka,2)/\lfloor \frac{a-16}{2} \rfloor \rfloor$.
18. For all $m \geqslant 3$ and $\ell \leqslant 2^{2m+1}$, $t(n,q,k,\ell) \geqslant \lfloor t(n,q,k(2^{m+1}+2^m-4),2)/2 \rfloor$.
19. For all $m \geqslant 4$ and $\ell \leqslant 2^{2m}$, $t(n,q,k,\ell) \geqslant \lfloor t(n,q,k(2^{m+1}-4),2)/2 \rfloor$.
20. For all $m \geqslant 1$ and $\ell \leqslant 2^{4m}$, $t(n,q,k,\ell) \geqslant \lfloor t(n,q,k(2^{2m+1}-2^m-1),2)/2 \rfloor$.
21. For all $m \geqslant 2$ and $\ell \leqslant 2^{4m+1}$, $t(n,q,k,\ell) \geqslant \lfloor t(n,q,k(2^{2m+1}+2^{2m}-2^m-2),2)/2 \rfloor$.
22. For all $m \geqslant 2$ and $\ell \leqslant 2^{4m+2}$, $t(n,q,k,\ell) \geqslant \lfloor t(n,q,k(2^{2^{2m+2}-2^m-2}),2)/2 \rfloor$.
23. For all $m \geqslant 2$ and $\ell \leqslant 2^{4m+3}$, $t(n,q,k,\ell) \geqslant \lfloor t(n,q,k(2^{2m+2}+2^{2m+1}-2^m-2),2)/2 \rfloor$.

Fig. 4. The relationship between floating codes with $\ell > 2$ and floating codes with $\ell = 2$. Here $t(n, q, k, \ell)$ denotes the optimal value of t (the number of rewrites) for a floating code with the parameters n, q, k, ℓ.

By using known results on covering codes [6], we can obtain a number of bounds for floating codes with large alphabets in terms of the performance of floating codes with binary alphabets. We report some of the results in Fig. 4.

To show how to derive the results in Fig. 4, we first need to define a few terms. Let $l(m, R)$ denote the smallest possible length of a binary linear code with codimension (i.e., redundancy) m and covering radius R. Let $t(n, k)$ denote the minimum possible covering radius of (n, k) binary linear codes. (Note that some of the letters here have different meanings from those used for floating codes. We use these notations following the convention of the study on covering codes [6].) A list of known results on binary linear covering codes are shown in Fig. 5.

We show how to derive the inequalities in Fig. 4 by two examples. The first example is the third inequality in Fig. 4: For $a \geqslant b \geqslant 1$ and $\ell \leqslant 2^{a-b}$

$$t(n, q, k, \ell) \geqslant \left\lfloor t(n, q, ka, 2)/\left\lceil \frac{a-b}{2} \right\rceil \right\rfloor.$$

By the third inequality in Fig. 5, when $a \geqslant b \geqslant 1$, $t[a, b] \leqslant \lceil \frac{a-b}{2} \rceil$. So if $\ell \leqslant 2^{a-b}$, we can map the variables of alphabet

1.	(*Hamming code:*) For $m \geqslant 2$, $l(m,1) = 2^m - 1$.
2.	(*Golay code:*) $l(11,3) = 23$.
3.	For $n \geqslant k \geqslant 1$, $t[n,k] \leqslant \lceil \frac{n-k}{2} \rceil$.
4.	For $k \geqslant 4$ and $n \geqslant 2^{k-2}$, $t[n,k] \leqslant \lfloor \frac{n}{2} \rfloor - 2^{(k-4)/2}$.
5.	$t(23,16) \leqslant 2$.
6.	$t(47,28) \leqslant 5$.
7.	For all $n \geqslant 1$, $t[n,1] = \lfloor \frac{n}{2} \rfloor$.
8.	For all $n \geqslant 2$, $t[n,2] = \lfloor \frac{n-1}{2} \rfloor$.
9.	For all $n \geqslant 3$, $t[n,3] = \lfloor \frac{n-2}{2} \rfloor$.
10.	For $n \geqslant 4$ and $n \neq 5$, $t[n,4] = \lfloor \frac{n-4}{2} \rfloor$.
11.	For $n \geqslant 5$ and $n \neq 6$, $t[n,5] = \lfloor \frac{n-5}{2} \rfloor$.
12.	For $n \geqslant 14$, $t[n,6] \leqslant \lfloor \frac{n-8}{2} \rfloor$.
13.	For $n \geqslant 19$, $t[n,7] \leqslant \lfloor \frac{n-9}{2} \rfloor$.
14.	For all $p \geqslant 2$ and for $n \geqslant 2^{2p} - 1$, $t[n,2p+1] \leqslant \lfloor \frac{n-2^p}{2} \rfloor$.
15.	For all $p \geqslant 2$ and for n even, $n \geqslant 2^{2p-1}$, $t[n,2p] \leqslant \lfloor \frac{n-2^{(2p-1)/2}}{2} \rfloor$.
16.	For all $p \geqslant 2$ and for n odd, $n \geqslant 2^{2p-1} - 1$, $t[n,2p] \leqslant \lfloor \frac{n-2^{(2p-1)/2}-1}{2} \rfloor$.
17.	For $n \geqslant 127$, $t[n,8] \leqslant \lfloor \frac{n-16}{2} \rfloor$.
18.	For all $m \geqslant 3$, $l(2m+1,2) \leqslant 2^{m+1} + 2^m - 4$.
19.	For all $m \geqslant 4$, $l(2m,2) \leqslant 2^{m+1} - 4$.
20.	For all $m \geqslant 1$, $l(4m,2) \leqslant 2^{2m+1} - 2^m - 1$.
21.	For all $m \geqslant 2$, $l(4m+1,2) \leqslant 2^{2m+1} + 2^{2m} - 2^m - 2$.
22.	For all $m \geqslant 2$, $l(4m+2,2) \leqslant 2^{2^{2m+2}-2^m-2}$.
23.	For all $m \geqslant 2$, $l(4m+3,2) \leqslant 2^{2m+2} + 2^{2m+1} - 2^m - 2$.

Fig. 5. Existing bounds for binary linear covering codes [6].

size ℓ to the cosets of a binary (a,b) linear covering code, whose covering radius is at most $\lceil \frac{a-b}{2} \rceil$. By Theorem 19, we get the third inequality of Fig. 4.

The second example is the 18th inequality in Fig. 4: For all $m \geqslant 3$ and $\ell \leqslant 2^{2m+1}$

$$t(n,q,k,\ell) \geqslant \lfloor t(n,q,k(2^{m+1}+2^m-4),2)/2 \rfloor.$$

By the 18th inequality in Fig. 5, when $m \geqslant 3$, $l(2m+1,2) \leqslant 2^{m+1} + 2^m - 4$. So when $m \geqslant 3$ and $\ell \leqslant 2^{2m+1}$, we can map variables of alphabet size ℓ to the cosets of a binary $(x, x-2m-1)$ linear covering code with covering radius 2, where $x = l(2m+1,2) \leqslant 2^{m+1} + 2^m - 4$. By Theorem 19, $t(n,q,k,\ell) \geqslant \lfloor t(n,q,kx,2)/2 \rfloor$. Since $x \leqslant 2^{m+1} + 2^m - 4$, $t(n,q,kx,2) \geqslant t(n,q,k(2^{m+1}+2^m-4),2)$. So we get the 18th inequality of Fig. 4.

The rest of the inequalities in Fig. 4 are derived similarly.

Since there have been a number of floating code constructions for binary variables, floating codes with large alphabets can also be built. The number of such results that can be obtained is large. We show some example data of the obtainable floating codes in Fig. 6.

V. Buffer Codes

In this section, we define buffer code and present its constructions. A buffer code is used for remembering the recent values in a data stream.

A. Definition

We formally define the buffer code as follows. Let v be a variable with alphabet $\{0,1,\ldots,\ell-1\}$, whose value is changed by rewrites as $u_1 \to u_2 \to u_3 \to \cdots$ over time. Let $r \geqslant 1$ be an integer parameter. The buffer code uses n q-ary cells to remember

n	q	k	ℓ	t	t_{up}	α
20	8	5	2	49	126	0.39
60	8	5	2	223	406	0.55
100	8	5	2	465	686	0.68
20	8	2	4	49	122	0.40
60	8	2	4	223	402	0.55
100	8	2	4	465	682	0.68
20	8	2	8	16	94	0.17
60	8	2	8	121	374	0.32
100	8	2	8	265	654	0.41
20	8	5	4	14	91	0.15
60	8	5	4	85	371	0.23
100	8	5	4	248	651	0.38

Fig. 6. Some example data on obtained floating codes. Here t is the number of rewrites guaranteed by the obtained code, t_{up} is an upper bound for t (computed using known results), and $\alpha = t/t_{up}$.

the most recent r values of v. That is, for $i = 1,2,3\ldots$, when the variable is rewritten as u_i, the buffer code can recover the vector $(u_{i-r+1}, u_{i-r+2}, \ldots, u_i)$. (By convention, we let $u_j = 0$ if $j \leqslant 0$.) With each rewrite, the cell levels can only increase, not decrease. A buffer code that supports t rewrites of the variable is called a (n,q,ℓ,r,t) buffer code. Our objective is to maximize t given the parameters n, q, ℓ, r.

Recording the last r values of a sequence is useful in practice for the implementation of certain data structures such as stacks. Buffer codes can also be used to record logged data in file/database systems, to checkpoint states, or to work as a buffer for data streaming applications.

Some examples of buffer codes have been shown in Fig. 2, where $n = 1$ and $\ell = 2$. Those codes can also be described by the state diagrams in Fig. 7. In the state diagrams, the r numbers inside a circle are the r recent variable values $(u_{i-r+1}, u_{i-r+2}, \ldots, u_i)$, and the number beside an edge shows by how much the cell level needs to increase for the corresponding rewrite. We now present a generalized construction for these codes.

B. A Single-Cell Construction

In this subsection, we present a $(n=1, q, \ell=2, r, t)$ buffer code and show that it achieves

$$t = \left\lfloor \frac{q}{2^{r-1}} \right\rfloor + r - 2.$$

In other words, the code allows $\lfloor \frac{q}{2^{r-1}} \rfloor + r - 2$ bits to be written into a q-ary cell. After every write, the last r bits written can be recovered.

The $(n=1, q, \ell=2, r, t)$ buffer code is defined by a surjective mapping, f_r, from $\mathbb{N}$ to $\{0,1\}^r$. The mapping f_r is defined recursively

$$f_1(x) = x \bmod 2$$
$$f_{r+1}(x) = \begin{cases} (0, \overline{f_r(x)}), & \text{if } x \bmod 2^{r+1} < 2^r \\ (1, f_r(x)), & \text{otherwise.} \end{cases}$$

Here $\overline{f_r(x)}$ is the negation of $f_r(x)$. That is, if we change all the bits in $f_r(x)$ from 1 to 0 and from 0 to 1, we get $\overline{f_r(x)}$. In the code, for $i = 0,1,\ldots q-1$, we let the cell level i represent the r variable values $(u_{i-r+1}, u_{i-r+2}, \ldots, u_i) = f_r(i)$.

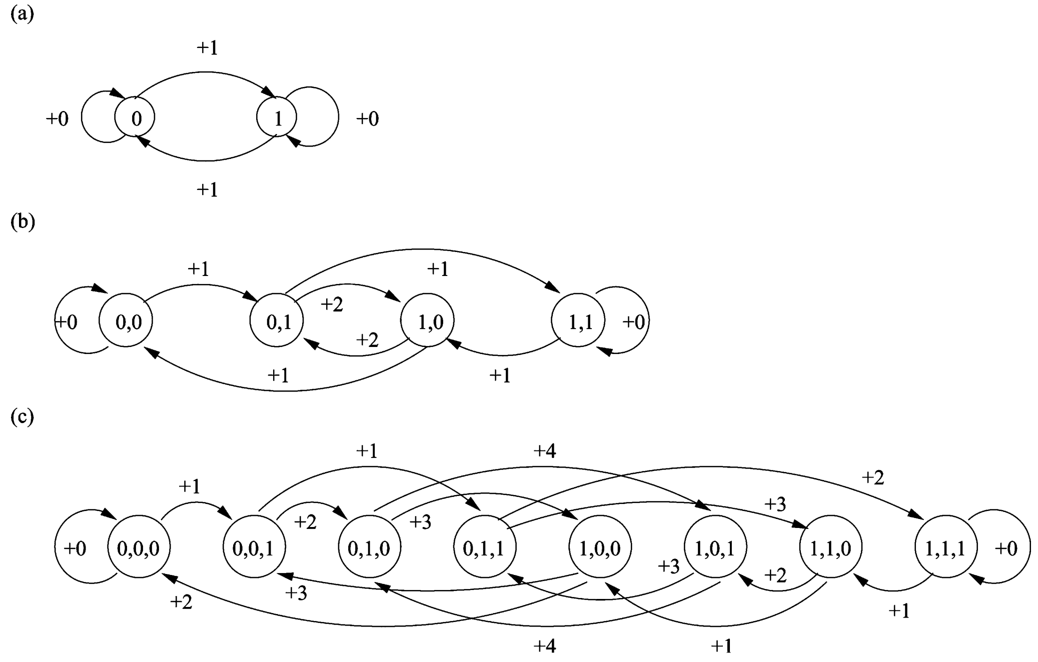

Fig. 7. State diagrams for the buffer codes with $n = 1$, $\ell = 2$. The r numbers inside a circle are the r recorded bits. The number beside an edge shows by how much the cell level needs to increase for the corresponding write. (a) $r = 1$. (b) $r = 2$. (c) $r = 3$.

We have shown examples of the code in Figs. 2 and 7. We now prove the performance of the code. Without loss of generality, let us assume $q \geqslant 2^r$.

Theorem 20: The $(n = 1, q, \ell = 2, r, t)$ buffer code supports

$$t = \left\lfloor \frac{q}{2^{r-1}} \right\rfloor + r - 2$$

rewrites.

Proof: We need to show that any binary sequence s of length at most t will bring the value of the cell level to at most $q-1$. We first show that the worst case sequence, s_t, is an alternation of 1 s and 0 s, namely the sequence

$$s_t = (1, 0, 1, 0, 1, 0 \ldots)$$

of length t. In the state diagram of Fig. 7(b), s_t corresponds to alternating states (0,1) and (1,0). Each state transition increases the cell level by 2, which is the maximum increase for this state diagram ($r = 2$). Similarly, in the $r = 3$ state diagram, shown in Fig. 7(c), s_t corresponds to alternating states (0,1,0) and (1,0,1), increasing the cell level by 4 for each bit written. In the general case, for the two vectors $(0, 1, \ldots, 0, 1)$ and $(1, 0, \ldots, 1, 0)$ (or $(0, 1, 0, 1, \ldots, 0)$ and $(1, 0, 1, 0, \ldots, 1)$), they are adjacent in the state diagram. What is more, the transition between them makes the cell level to increase by 2^{r-1}, which happens to be the largest possible increment of the cell level for a write. Therefore, it is not hard to verify that the sequence s_t is the worst case sequence. The $r-1$ initial writes increase the cell level respectively by $2^0, 2^1, \ldots, 2^{r-2}$, after which the increment is 2^{r-1}. Each of the increments is the maximum possible. Therefore

$$t = \left\lfloor \frac{q}{2^{r-1}} \right\rfloor + r - 2.$$

■

We now present an upper bound to t for buffer codes with $n = 1$ and arbitrary ℓ, q, r.

Theorem 21: When $n = 1$, every buffer code has

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

Proof: Since r writes can completely change the r recent variable values, there is always a sequence of r writes that increases the cell level by at least $\ell^r - 1$. We choose the first set of r writes, the second set of r writes, …, the bth set of r writes such that every such set of r writes increases the cell level by at least $\ell^r - 1$. Let b be as large as possible. After those br writes, select a set of y writes after which no more write can be performed. Let y be as small as possible. Clearly, $y < r$.

Since the maximum cell level is $q-1$, $b \leqslant \left\lfloor \frac{q-1}{\ell^r-1} \right\rfloor$. Note that $\lfloor \log_\ell \{[(q-1) mod(\ell^r - 1)] + 1\} \rfloor < r$. If $b < \lfloor \frac{q-1}{\ell^r-1} \rfloor$, then $t \leqslant br + y < \lfloor \frac{q-1}{\ell^r-1} \rfloor \cdot r \leqslant \lfloor \frac{q-1}{\ell^r-1} \rfloor \cdot r + \lfloor \log_\ell \{[(q-1) mod(\ell^r-1)] + 1\} \rfloor$. Now consider the case that $b = \left\lfloor \frac{q-1}{\ell^r-1} \right\rfloor$. In that case, the last y writes increase the cell's level by at most $(q-1) mod(\ell^r - 1)$. As y or fewer writes lead the variable value to ℓ^y possible values, with the same analysis as before, we get $\ell^y - 1 \leqslant (q-1) mod(\ell^r - 1)$. So $y \leqslant \lfloor \log_\ell \{[(q-1) mod(\ell^r - 1)] + 1\} \rfloor$. So again, $t \leqslant br + y \leqslant \lfloor \frac{q-1}{\ell^r-1} \rfloor \cdot r + \lfloor \log_\ell \{[(q-1) mod(\ell^r - 1)] + 1\} \rfloor$. So the theorem holds. ■

C. Code Construction for $\ell = 2$ and General n, q, r

We present a buffer code for $\ell = 2$ and general n, q, r with $n \geqslant 2r$. The code achieves $t = (q-1)(n-2r+1) + r - 1$.

We first define some terms that will be used in the rest of the paper. For $1 \leqslant i \leqslant n$, we use a_i to denote the ith cell. We use $(c_1, c_2, \ldots, c_n)$—called the cell-state vector—to denote the states of the n cells, where $c_i (0 \leqslant c_i \leqslant q-1)$ is the state of the

ith cell. We use $(v_1, v_2, \dots, v_r)$—called the variable vector—to denote the most recent r values of v, with v_r being the most recent value and v_1 being the oldest among the most recent r values. That is, when the value u_i is written, $v_1 = u_{i-r+1}$, $v_2 = u_{i-r+2}$, $\cdots$, and $v_r = u_i$. We define the cell-state vectors of the ith generation $(0 \leqslant i \leqslant t)$ to be the set of cell-state vectors reachable after exactly i rewrites since the beginning.

We first present the buffer code construction for the special case $q = 2$. We will then naturally extend the code construction for arbitrary q. The code is defined as follows.

- Mapping cell-state vectors to variable vectors: By valid cell-state vector, we mean a cell state vector that can be reached by some rewrite operations. Every valid cell-state vector $(c_1, c_2, \dots, c_n)$ of this code satisfies the following property: For $i = 1, 2, \dots, n-r$, for any cell-state vector of the ith generation, there are exactly i cells in the state 1 and $n-i$ cells in the state 0; what's more, all those i cells in the state 1 belong to the set $\{a_1, a_2, \dots, a_{i+r}\}$ (namely, the first $i+r$ cells).
  Clearly, a valid cell state vector $(c_1, c_2, \dots, c_n)$ is in the $(\sum_{i=1}^{n} c_i)$th generation.
  A valid cell state vector $(c_1, c_2, \dots, c_n)$ in the ith generation is mapped to the variable vector $(v_1, v_2, \dots, v_r)$ as follows: For $j = 1, 2, \dots, r$, $v_j = c_{i+j}$.
- *Writing:* The code enables $n-r$ rewrites. Let's say that the current cell state vector $(c_1, c_2, \dots, c_n)$ is $(x_1, x_2, \dots, x_n)$ and it is in the ith generation. $(0 \leqslant i < n-r.)$ Say that the next rewrite is to change the variable's value to y. (By default, only the rewrites that change the variable vector are considered. It means that if the current variable vector is $(0, 0, \dots, 0)$ or $(1, 1, \dots, 1)$, then y cannot be 0 or 1, respectively.) Then, if $y = 0$, find an integer $j \leqslant i+1$ such that $x_j = 0$, and change c_j—the state of the jth cell—to be 1; if $y = 1$, then change c_{i+r+1} from 0 to 1.

The following is an example of the code.

Example 22: Let $\ell = 2$, $n = 9$, $q = 2$, and $r = 3$. If the $n - r = 6$ rewrites change the variable vector as $(0,0,0) \to (0,0,1) \to (0,1,1) \to (1,1,0) \to (1,0,0) \to (0,0,1) \to (0,1,0)$, then the cell-state vector changes as $(0,0,0,0,0,0,0,0,0) \to (0,0,0,1,0,0,0,0,0) \to (0,0,0,1,1,0,0,0,0) \to (0,0,1,1,1,0,0,0,0) \to (0,1,1,1,1,0,0,0,0) \to (0,1,1,1,1,0,0,1,0) \to (0,1,1,1,1,1,0,1,0)$. We can see that given a cell-state vector, recovering the variable vector is very simple: just read the $(w+1)$th, $(w+2)$th, $\cdots$, $(w+r)$th entries in the cell state vector, where w is the number of 1's in the vector.

We now extend the above code from $q = 2$ to $q \geqslant 2$ by using the "level by level" approach: first use levels 0 and 1, then levels 1 and 2, $\cdots$, and finally levels $q-2$ and $q-1$ to encode the data. Note that when the levels 0, 1 are used, the code can support $n - r$ rewrites. Then for $i = 1, \dots, q-2$, when we transit from levels $i-1$, i to levels i, $i+1$, we may need to set as many as r cell levels to $i+1$ for the first rewrite; so the levels i, $i+1$ can support $n - 2r + 1$ rewrites.

Theorem 23: The buffer code presented above is valid, and it supports $t = (q-1)(n-2r+1) + r - 1$ rewrites.

Proof: First, assume $q = 2$. To prove the code is valid, we use induction to prove the following assertion: "For $i = 1, 2, \dots, n-r$, the ith rewrite leads the cells to a valid cell state vector that correctly corresponds to the new variable vector." Consider the case $i = 1$. The first rewrite has only one possibility: to change the variable to 1. By the code construction, the cell state $(c_1, c_2, \dots, c_n)$ becomes as follows: $c_{r+1} = 1$, and $c_j = 0$ for all $j \neq r+1$. That cell state is valid and corresponds to the variable vector $(0, 0, \dots, 0, 1)$. So the assertion holds when $i = 1$. That serves as the base case.

Assume that the *assertion* holds for all $i < p$, where $p \leqslant n - r$. Now consider the case $i = p$. Say that the pth rewrite changes the variable to y, where $y = 0$ or 1. By the induction assumption, after the $(p-1)$th rewrite, $p-1$ cells are in the state 1, and they all belong to the first $p - 1 + r$ cells (namely, cells $a_1, a_2, \dots, a_{p-1+r}$); therefore, among the first p cells, at least one of them is in state 0. If $y = 0$, the pth rewrite changes such a cell in state 0 to state 1, so the number of cells in state 1 becomes p; if $y = 1$, the $(p+r)$th cell is changed from 0 to 1, so the number of cells in state 1 also becomes p. Clearly, after the pth rewrite, all those cells in state 1 are among the first $p + r$ cells. Therefore, the cell state vector after the pth rewrite is *valid*. Say that after the $(p-1)$th rewrite, the cell state vector is $(c_1, c_2, \dots, c_n)$. Its corresponding variable vector is simply $(v_1 = c_p, v_2 = c_{p+1}, \dots, v_r = c_{p+r-1})$. After the pth rewrite, the state of the $(p+r)$th cell becomes y, so the corresponding variable vector is $(v_1 = c_{p+1}, v_2 = c_{p+2}, \dots, v_{r-1} = c_{p+r-1}, v_r = y)$, which is the correct variable vector. So the assertion holds when $i = p$. This completes the induction for $q = 2$. When $q \geqslant 2$, the code uses the "level by level" approach, and it is simple to see that the conclusion holds. ■

D. Enhanced Buffer Code for $\ell = 2$, $r = 2$ and General n, q

The code presented in the previous subsection has a t that is asymptotically optimal in n, q (for $r = o(n)$). When $r = 2$, it gives $t = (q-1)(n-3) + 1$. In this section, we present a better code with $t = (q-1)(n-1)$. In particular, when $q = 2$, this code is strictly optimal.

We first present the new code construction for the case $q = 2$, and analyze its properties. The construction is then extended for general q using the "level by level" approach.

The new buffer code enhances the code construction of the previous subsection. When $q = 2$, it has $t = n - 1$. The new code uses the same method as the previous code to map cell-state vectors of the 1st, 2nd, $\cdots$, $(n-2)$th generations to variable vectors. It adds the following specification to the previous code construction to handle the first $n - 2$ rewrites:

- *Writing:* Let's say that the current cell-state vector $(c_1, c_2, \dots, c_n)$ is $(x_1, x_2, \dots, x_n)$ and it is in the ith generation, where $0 \leqslant i < n-2$. (The corresponding variable vector is $(v_1 = x_{i+1}, v_2 = x_{i+2})$.) Say that the next rewrite is to change the variable's value to y. The rewrite is performed as follows:
  1) If $y = 0$ and $(x_{i+1}, x_{i+2}) = (0, 1)$, then change c_{i+1}—the state of the $(i+1)$th cell—to 1.
  2) If $y = 0$ and $(x_{i+1}, x_{i+2}) = (1, 0)$, then find the integer $j \leqslant i$ such that $x_j = 0$, and change c_j to 1.

3) If $y = 0$ and $(x_{i+1}, x_{i+2}) = (1,1)$, then find the integer $j \leqslant i$ such that "$x_j = 0$ and $(i+3) - j$ is an even integer", and change c_j to 1.
4) If $y = 1$, change c_{i+3} from 0 to 1.

The mapping from the cell-state vectors in the $(n-1)$th generation to the variable vectors is as follows:

- Mapping from cell-state vectors to variable vectors: Every valid cell-state vector in the $(n-1)$th generation satisfies this property: Among the n cells, $n-1$ of them are in state 1 and one of them is in state 0.
  Given a valid cell state vector in the $(n-1)$th generation, let's say that a_i—the ith cell—is the unique cell in state 0. The cell state vector is mapped to the variable vector (v_1, v_2) in the following way:
  1) If $i \leqslant n-2$ and $n-i$ is even, then $(v_1, v_2) = (1,0)$.
  2) If $i \leqslant n-2$ and $n-i$ is odd, then $(v_1, v_2) = (0,0)$.
  3) If $i = n-1$, then $(v_1, v_2) = (1,1)$.
  4) If $i = n$, then $(v_1, v_2) = (0,1)$.

The $(n-1)$th rewrite is performed in the following way:

- The $(n-1)$th rewrite: Let's say that after the $(n-2)$th rewrite, the cell-state vector $(c_1, c_2, \ldots, c_n)$ is $(x_1, x_2, \ldots, x_n)$. (The corresponding variable vector is $(v_1 = x_{n-1}, v_2 = x_n)$.) Say that the $(n-1)$th rewrite is to change the variable's value to y. It is performed as follows:
  1) If "$y = 0$ and $(x_{n-1}, x_n) = (0,1)$" or "$y = 1$ and $(x_{n-1}, x_n) = (0,0)$," then change c_{n-1} from 0 to 1.
  2) If $y = 0$ and $(x_{n-1}, x_n) = (1,0)$, then change c_n from 0 to 1.
  3) If $y = 0$ and $(x_{n-1}, x_n) = (1,1)$, then let $j \leqslant n-2$ be the integer such that "$x_j = 0$ and $n-j$ is odd", and change c_j from 0 to 1.
  4) If $y = 1$ and $(x_{n-1}, x_n) = (0,1)$ or (1,0), then let $j \leqslant n-2$ be the integer such that $x_j = 0$, and change c_j from 0 to 1.

Example 24: Let $\ell = 2$, $n = 6$, $q = 2$, $r = 2$. If the $n - 1 = 5$ rewrites change the variable vector as $(0,0) \to (0,1) \to (1,0) \to (0,1) \to (1,1) \to (1,0)$, then the cell-state vector changes as $(0,0,0,0,0,0) \to (0,0,1,0,0,0) \to (0,1,1,0,0,0) \to (0,1,1,0,1,0) \to (0,1,1,0,1,1) \to (1,1,1,0,1,1)$.

The next lemma shows a special property of the new code.

Lemma 25: For the new code constructed in this subsection, for $i = 0, 1, \ldots, n-2$, let $(c_1, c_2, \ldots, c_n)$ be a valid cell-state vector in the ith generation. By the code construction, among the first $i+2$ cells—$a_1, a_2, \ldots, a_{i+2}$—exactly two of them are in the state 0. Let a_p and a_q be those two cells. Then, between p and q, one is odd and the other is even.

Proof: The proof is by induction on i. When $i = 0$, $c_1 = c_2 = 0$, so p=1 and $q = 2$. So the lemma holds when $i = 0$. This serves as the base case.

Assume that when $i < z \leqslant n-2$, the lemma holds. Now consider the case $i = z$. The proof for this induction step is a straightforward check using the rule on writing in the code construction. For example, consider the following case: after $z-1$ writes, the states of a_z and a_{z+1} are 0 and 1, respectively, and the zth write changes the variable to 0. In this case, the code construction changes the state of a_z to 1. By the induction assumption, after $z-1$ writes, there is a cell $a_j (j \leqslant z+1)$ whose state is 0 such that between j and z, one is odd and one is even. After the zth write, both a_j and a_{z+2} are in the state 0, so we can let $p = j$ and $q = z+2$; then between p and q, one is odd and the other is even; so the lemma holds. All the other cases can be checked similarly; for simplicity, we skip the details. That completes the induction. So the lemma holds for all $0 \leqslant i \leqslant n-2$. ■

Theorem 26: The new buffer code constructed in this subsection is valid. And it supports $t = n-1$ rewrites, which is optimal.

Proof: It is easy to verify that the new code deals with the first $n-2$ writes and the 0th, 1st, $\ldots, (n-2)$th generations of cell state vectors in the same way as the code construction in the previous subsection does, except that the $n-2$ writes are performed in a more specific way. For succinctness, we omit the details of this simple verification. Now consider the $(n-1)$th write. Based on Lemma 25, any cell state vector in the $(n-2)$th generation has exactly two cells a_p, a_q whose states are 0, while between p and q one is odd and the other is even. By using this observation, and by the way the code construction performs the $(n-1)$th write and maps the $(n-1)$th generation of cell-state vectors to variable vectors, we can easily use a case by case verification to see that the $(n-1)$th write always leads the cells to a valid cell-state vector that corresponds to the correct variable vector. So the code is correct. It directly follows from the code construction that $t = n-1$. ■

The above code construction and analysis are for $q = 2$. When $q \geqslant 2$, we can use the cells "level by level" in the same way as before. For such a code, t becomes $(q-1)(n-2)+1$.

VI. Conclusion

With the wide application of flash memories, it has become important to design appropriate coding schemes for them. For flash memories, due to the high cost of block erasures, it is a critical requirement to rewrite data efficiently. Different from the current techniques used in flash memories, such as wear leveling for balancing erasures, rewriting codes can minimize the total number of erasures. In this paper, we focus on the joint storage of data, with the objective of maximizing the number of times the data can be rewritten. We define floating codes and buffer codes, two rewriting codes designed for different applications. We explore the information theoretic bounds for the two codes, and present a set of code constructions. The results show that the rewriting capabilities of different data variables can be integrated to a high degree.

Acknowledgment

The authors would like to thank the anonymous reviewers for their helpful and constructive suggestions.

References

[1] R. Ahlswede and N. Cai, "Models of multi-user write-efficient memories and general diametric theorems," *Inf. Computat.*, vol. 135, no. 1, pp. 37–67, 1997.
[2] R. Ahlswede and Z. Zhang, "Coding for write-efficient memory," *Inf. Computat.*, vol. 83, no. 1, pp. 80–97, 1989.

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

[4] *Flash Memories*, P. Cappelletti, C. Golla, P. Olivo, and E. Zanoni, Eds., 1st ed. Boston, MA: Kluwer, 1999.

[5] G. Cohen, P. Godlewski, and F. Merkx, "Linear binary code for write-once memories," *IEEE Trans. Inf. Theory*, vol. IT-32, pp. 697–700, Sep. 1986.

[6] G. Cohen, I. Honkala, S. Litsyn, and A. Lobstein, *Covering Codes*. Amsterdam, The Netherlands: North-Holland/Elsevier, 1997.

[7] A. Fiat and A. Shamir, "Generalized "write-once" memories," *IEEE Trans. Inf. Theory*, vol. IT-30, pp. 470–480, May 1984.

[8] H. Finucane, Z. Liu, and M. Mitzenmacher, "Designing floating codes for expected performance," in *Proc. 46th Ann. Allerton Conf. Commun., Contr., Comput.*, Monticello, IL, Sep. 23–26, 2008, pp. 1389–1396.

[9] F. Fu and A. J. Han Vinck, "On the capacity of generalized write-once memory with state transitions described by an arbitrary directed acyclic graph," *IEEE Trans. Inf. Theory*, vol. 45, pp. 308–313, 1999.

[10] F. Fu and R. Yeung, "On the capacity and error-correcting codes of write-efficient memories," *IEEE Trans. Inf. Theory*, vol. 46, pp. 2299–2314, 2000.

[11] E. Gal and S. Toledo, "Algorithms and data structures for flash memories," *ACM Comput. Surveys*, vol. 37, no. 2, pp. 138–163, 2005.

[12] C. Heegard, "On the capacity of permanent memory," *IEEE Trans. Inf. Theory*, vol. IT-31, pp. 34–42, Jan. 1985.

[13] C. Heegard and A. A. El Gamal, "On the capacity of computer memory with defects," *IEEE Trans. Inf. Theory*, vol. IT-29, pp. 731–739, Sep. 1983.

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

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

[16] A. Jiang and J. Bruck, Joint Coding for Flash Memory Storage, Caltech Tech. Rep. [Online]. Available: http://www.paradise.caltech.edu/etr.html

[17] A. Jiang, M. Langberg, M. Schwartz, and J. Bruck, "Universal rewriting in constrained memories," in *Proc. IEEE Int. Symp. Inf. Theory (ISIT)*, Seoul, Korea, Jun.–Jul. 2009, pp. 1219–1223.

[18] A. Jiang, R. Mateescu, M. Schwartz, and J. Bruck, "Rank modulation for flash memories," *IEEE Trans. Inf. Theory*, vol. 55, pp. 2659–2673, 2009.

[19] A. V. Kuznetsov and B. S. Tsybakov, "Coding for memories with defective cells," *Problemy Peredachi Informatsii*, vol. 10, no. 2, pp. 52–60, 1974.

[20] A. V. Kuznetsov and A. J. Han Vinck, "On the general defective channel with informed encoder and capacities of some constrained memories," *IEEE Trans. Inf. Theory*, vol. 40, pp. 1866–1871, Nov. 1994.

[21] L. A. Lastras-Montano, M. Franceschini, T. Mittelholzer, J. Karidis, and M. Wegman, "On the lifetime of multilevel memories," in *Proc. IEEE Int. Symp. Inf. Theory (ISIT)*, Seoul, Korea, 2009, pp. 1224–1228.

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

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

[24] W. M. C. J. van Overveld, "The four cases of write unidirectional memory codes over arbitrary alphabets," *IEEE Trans. Inf. Theory*, vol. 37, pp. 872–878, 1991.

[25] R. Rivest and A. Shamir, "How to reuse a "write-once" memory," *Inf. Control*, vol. 55, pp. 1–19, 1982.

[26] G. Simonyi, "On write-unidirectional memory codes," *IEEE Trans. Inf. Theory*, vol. 35, pp. 663–667, May 1989.

[27] F. M. J. Willems and A. J. Han Vinck, "Repeated recording for an optical disk," in *Proc. 7th Symp. Inf. Theory in the Benelux*, May 22–23, 1986, pp. 49–53.

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

[29] 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, poster presented at ISIT.

[30] E. Yaakobi, A. Vardy, P. H. Siegel, and J. K. Wolf, "Multidimensional flash codes," in *Proc. 46th Ann. Allerton Conf. Commun., Contr., Comput.*, Monticello, IL, Sep. 23–26, 2008, pp. 392–399.

[31] G. Zémor and G. Cohen, "Error-correcting WOM-codes," *IEEE Trans. Inf. Theory*, vol. 37, pp. 730–734, May 1991.

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

He is currently an Assistant Professor in the Computer Science and Engineering Department, Texas A&M University, College Station.

Prof. Jiang is a recipient of the NSF CAREER Award in 2008 for his research on information theory for flash memories. His research interests include information theory, data storage, networks, and algorithm design.

Vasken Bohossian received the B.S.E. degree in electrical engineering (honors program) from McGill University, Canada, in 1993, the M.S. degree in electrical engineering, and the Ph.D. degree in computation and neural systems from the California Institute of Technology, Pasadena, in 1994 and 1998, respectively.

He is a cofounder of Rainfinity, acquired by EMC Corporation in 2005. His research interests include parallel and distributed computing, fault-tolerant computing, error-correcting codes, computation theory, and threshold logic.

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, in 1982 and 1985, respectively, and the Ph.D. degree in electrical engineering from Stanford University, Stanford, CA, in 1989.

His research combines work on information theory and systems and the theoretical study of computation in biological circuits. He has extensive industrial experience. He was with IBM Research where he participated in the design and implementation of the first IBM parallel computer. He was a co-founder and chairman of Rainfinity, a spin-off company from Caltech that focused on software products for management of network information storage systems.

Dr. Bruck is a recipient of the Feynman Prize for Excellence in Teaching, a Sloan Research Fellowship, a National Science Foundation Young Investigator Award, an IBM Outstanding Innovation Award, an IBM Outstanding Technical Achievement Award, and the IEEE Schelkunoff Award.