Rank Modulation for Flash Memories

Anxiao (Andrew) Jiang, *Member, IEEE*, Robert Mateescu, *Member, IEEE*, Moshe Schwartz, *Member, IEEE*, and Jehoshua Bruck, *Fellow, IEEE*

***Abstract*—We explore a novel data representation scheme for multilevel flash memory cells, in which a set of n cells stores information in the permutation induced by the different charge levels of the individual cells. The only allowed charge-placement mechanism is a "push-to-the-top" operation, which takes a single cell of the set and makes it the top-charged cell. The resulting scheme eliminates the need for discrete cell levels, as well as overshoot errors, when programming cells.**

We present unrestricted Gray codes spanning all possible n-cell states and using only "push-to-the-top" operations, and also construct balanced Gray codes. One important application of the Gray codes is the realization of logic multilevel cells, which is useful in conventional storage solutions. We also investigate rewriting schemes for random data modification. We present both an optimal scheme for the worst case rewrite performance and an approximation scheme for the average-case rewrite performance.

***Index Terms*—Asymmetric channel, flash memory, Gray codes, permutations, rank modulation.**

I. Introduction

FLASH memory is a nonvolatile memory both electrically programmable and electrically erasable. Its reliability, high storage density, and relatively low cost have made it a dominant nonvolatile memory technology and a prominent candidate to replace the well-established magnetic recording technology in the near future.

The most conspicuous property of flash storage is its inherent asymmetry between cell programming (charge placement) and cell erasing (charge removal). While adding charge to a single cell is a fast and simple operation, removing charge from a single cell is very difficult. In fact, today, most (if not all) flash memory technologies do not allow a single cell to be erased but rather only a large block of cells. Thus, a single-cell erase operation requires the cumbersome process of copying an entire block to a temporary location, erasing it, and then programming all the cells in the block.

Manuscript received September 18, 2008; revised January 28, 2009. Current version published May 20, 2009. This work was supported in part by the Caltech Lee Center for Advanced Networking, by the National Science Foundation (NSF) under Grant ECCS-0802107 and the NSF CAREER Award 0747415 , by the GIF under Grant 2179-1785.10/2007, by the NSF-NRI, and by a gift from Ross Brown. The material in this paper was presented in part at the IEEE International Symposium on Information Theory, Toronto, ON, Canada, July 2008

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

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

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).

Communicated by T. Etzion, Associate Editor for Coding Theory.

Digital Object Identifier 10.1109/TIT.2009.2018336

To keep up with the ever-growing demand for denser storage, the multilevel flash cell concept is used to increase the number of stored bits in a cell [8]. Instead of the usual single-bit flash memories, where each cell is in one of two states (erased/programmed), each multilevel flash cell stores one of q levels and can be regarded as a symbol over a discrete alphabet of size q. This is done by designing an appropriate set of *threshold levels* which are used to quantize the charge level readings to symbols from the discrete alphabet.

Fast and accurate programming schemes for multilevel flash memories are a topic of significant research and design efforts [2], [14], [31]. All these and other works share the attempt to iteratively program a cell to an exact prescribed charge level in a minimal number of programming cycles. As mentioned above, flash memory technology does not support charge removal from individual cells. As a result, the programming cycle sequence is designed to cautiously approach the target charge level from below so as to avoid undesired global erases in case of overshoots. Consequently, these attempts still require many programming cycles, and they work only up to a moderate number of levels per cell.

In addition to the need for accurate programming, the move to multilevel flash cells also aggravates reliability. The same reliability aspects that have been successfully handled in single-level flash memories may become more pronounced and translate into higher error rates in stored data. One such relevant example is errors that originate from low *memory endurance* [5], by which a drift of threshold levels in aging devices may cause programming and read errors.

We therefore propose the *rank-modulation* scheme, whose aim is to eliminate both the problem of overshooting while programming cells, and the problem of memory endurance in aging devices. In this scheme, an ordered set of n cells stores the information in the permutation induced by the charge levels of the cells. In this way, no discrete levels are needed (i.e., no need for threshold levels) and only a basic charge-comparing operation (which is easy to implement) is required to read the permutation. If we further assume that the only programming operation allowed is raising the charge level of one of the cells above the current highest one (*push-to-the-top*), then the overshoot problem is no longer relevant. Additionally, the technology may allow in the near future the decrease of all the charge levels in a block of cells by a constant amount smaller than the lowest charge level (*block deflation*), which would maintain their relative values, and thus leave the information unchanged. This can eliminate a designated erase step, by deflating the entire block whenever the memory is not in use.

Once a new data representation is defined, several tools are required to make it useful. In this paper, we present Gray codes that bring to bear the full representational power of rank mod-

0018-9448/$25.00 © 2009 IEEE

ulation, and data rewriting schemes. The Gray code [13] is an ordered list of distinct length n binary vectors such that every two adjacent words (in the list) differ by exactly one bit flip. They have since been generalized in countless ways and may now be defined as an ordered set of distinct states for which every state s_i is followed by a state s_{i+1} such that $s_{i+1} = t(s_i)$, where $t \in T$ is a *transition function* from a predetermined set T defining the Gray code. In the original code, T is the set of all possible single bit flips. Usually, the set T consists of transitions that are minimal with respect to some cost function, thus creating a traversal of the state space that is minimal in total cost. For a comprehensive survey of combinatorial Gray codes, the reader is referred to [33].

One application of the Gray codes is the realization of logic multilevel cells with rank modulation. The traversal of states by the Gray code is mapped to the increase of the cell level in a classic multilevel flash cell. In this way, rank modulation can be naturally combined with current multilevel storage solutions. Some of the Gray code constructions we describe also induce a simple algorithm for generating the list of permutations. Efficient generation of permutations has been the subject of much research as described in the general survey [33], and the more specific [34] (and references therein). In [34], the transitions we use in this paper are called "nested cycling," and the algorithms cited there produce lists that are not Gray codes since some of the permutations repeat, which makes the algorithms inefficient.

We also investigate efficient rewriting schemes for rank modulation. Since it is costly to erase and reprogram cells, we try to maximize the number of times data can be rewritten between two erase operations [4], [21], [22]. For rank modulation, the key is to minimize the highest charge level of cells. We present two rewriting schemes that are, respectively, optimized for the worst case and the average-case performance.

Rank modulation is a new storage scheme and differs from existing data storage techniques. There has been some recent work on coding for flash memories. Examples include *floating codes* [22], [23], which jointly record and rewrite multiple variables, and *buffer codes* [4], [37], that keep a log of the recent modifications of data. Both floating codes and buffer codes use the flash cells in a conventional way, namely, the fixed discrete cell levels. Floating codes are an extension of the *write-once memory* (WOM) codes [6], [10], [11], [17], [32], [36], which are codes for effective rewriting of a single variable stored in cells that have irreversible state transitions. The study in this area also includes defective memories [16], [18], where defects (such as "stuck-at faults") randomly happen to memory cells and how to store the maximum amount of information is considered. In all the above codes, unlike rank modulation, the states of different cells do not relate to each other. Also related is the work on *permutation codes* [3], [35], used for data transmission or signal quantization.

The paper is organized as follows: Section II describes a Gray code that is cyclic and complete (i.e., it spans the entire symmetric group of permutations); Section III introduces a Gray code that is cyclic, complete and balanced, optimizing the transition step and also making it suitable for block deflation; Section IV shows a rewriting scheme that is optimal for the worst case rewrite cost; Section V presents a code optimized for the average rewrite cost with small approximation ratios; Section VI concludes this paper.

II. Definitions and Basic Construction

Let S be a *state space*, and let T be a set of *transition functions*, where every $t \in T$ is a function $t : S \to S$. A *Gray code* is an ordered list $s_1, s_2, \ldots, s_m$ of distinct elements from S such that for every $1 \leqslant i \leqslant m-1$, $s_{i+1} = t(s_i)$ for some $t \in T$. If $s_1 = t(s_m)$ for some $t \in T$, then the code is *cyclic*. If the code spans the entire space S we call it *complete*.

Let $[n]$ denote the set of integers $\{1, 2, \ldots, n\}$. An ordered set of n flash memory cells named $1, 2, \ldots, n$, each containing a distinct charge level, induces a permutation of $[n]$ by writing the cell names in descending charge level $[a_1, a_2, \ldots, a_n]$, i.e., the cell a_1 has the highest charge level while a_n has the lowest. The state space for the rank modulation scheme is therefore the set of all permutations over $[n]$, denoted by S_n.

As described in the previous section, the basic minimal-cost operation on a given state is a "push-to-the-top" operation by which a single cell has its charge level increased so as to be the highest of the set. Thus, for our basic construction, the set T of minimal-cost transitions between states consists of $n-1$ functions pushing the ith element of the permutation, $2 \leqslant i \leqslant n$, to the front

$$t_i([a_1, \ldots, a_{i-1}, \boldsymbol{a_i}, a_{i+1}, \ldots, a_n]) = [\boldsymbol{a_i}, a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n].$$

Throughout this work, our state space S will be the set of permutations over $[n]$, and our set of transition functions will be the set T of "push-to-the-top" functions. We call such a code a length-n rank modulation Gray code (n-RMGC).

Example 1: An example of a 3-RMGC is the following:

$$\begin{array}{cccccc} 1 & 2 & 3 & 1 & 3 & 2 \\ 2 & 1 & 2 & 3 & 1 & 3 \\ 3 & 3 & 1 & 2 & 2 & 1 \end{array}$$

where the permutations are the columns being read from left to right. The sequence of operations creating this cyclic code is: $t_2, t_3, t_3, t_2, t_3, t_3$. This sequence will obviously create a Gray code regardless of the choice of the first column.

One important application of the Gray codes is the realization of logic multilevel cells. The traversal of states by the Gray code is mapped to the increase of the cell level in a classic multilevel flash cell. As an n-RMGC has $n!$ states, it can simulate a cell of up to $n!$ discrete levels. Current data storage schemes (e.g., floating codes [22]) can therefore use the Gray codes as logic cells, as illustrated in Fig. 1, and get the benefits of rank modulation.

We will now show a basic recursive construction for n-RMGCs. The resulting codes are *cyclic* and *complete*, in the sense that they span the entire state space. Our recursion basis is the simple 2-RMGC: $[1, 2]$, $[2, 1]$.

Now let us assume we have a cyclic and complete $(n-1)$-RMGC, which we call C_{n-1}, defined by the sequence of transitions $t_{i_1}, t_{i_2}, \ldots, t_{i_{(n-1)!}}$ and where $t_{i_{(n-1)!}} = t_2$, i.e., a "push-to-the-top" operation on the second element in the

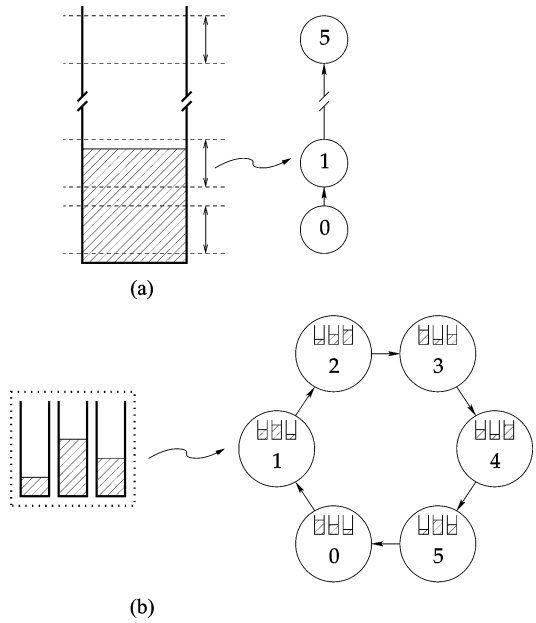

Fig. 1. Two multilevel flash-memory cells with six levels, currently storing the value "1." (a) The first is realized using a single multilevel cell with absolute thresholds. The possible transitions between states are shown to its right. (b) The second is realized by combining three flash cells with no thresholds and by using a rank-modulation scheme. The possible transitions between states are given by the 3-RMGC of Example 1.

permutation.[1] We further assume that the transition t_2 appears at least twice. We will now show how to construct C_n, a cyclic and complete n-RMGC with the same property.

We set the first permutation of the code to be $[1, 2, \ldots, n]$, and then use the transitions $t_{i_1}, t_{i_2}, \ldots, t_{i_{(n-1)!-1}}$ to get a list of $(n-1)!$ permutations which we call the first *block* of the construction. By our assumption, the permutations in this list are all distinct, and they all share the property that their last element is n (since all the transitions use just the first $n-1$ elements). Furthermore, since $t_{i_{(n-1)!}} = t_2$, we know that the last permutation generated so far is $[2, 1, 3, \ldots, n-1, n]$.

We now use t_n to create the first permutation of the second block of the construction, and then use $t_{i_1}, t_{i_2}, \ldots, t_{i_{(n-1)!-1}}$ again to create the entire second block. We repeat this process $n-1$ times, i.e., use the sequence of transitions $t_{i_1}, t_{i_2}, \ldots, t_{i_{(n-1)!-1}}, t_n$ a total of $n-1$ times to construct $n-1$ blocks, each containing $(n-1)!$ permutations.

The following two simple lemmas extend the intuition given above.

[1]This last requirement merely restricts us to have t_2 used *somewhere* since we can always rotate the set of transitions to make t_2 be the last one used.

Lemma 2: The second element in the first permutation in every block is 2. The first element in the last permutation in every block is also 2.

Proof: During the construction process, in each block we use the transitions $t_{i_1}, t_{i_2}, \ldots, t_{i_{(n-1)!-1}}$ in order. If we were to use the transition $t_{i_{(n-1)!}} = t_2$ next, we would return to the first permutation of the block since $t_{i_1}, t_{i_2}, \ldots, t_{i_{(n-1)!}}$ are the transitions of a cyclic $(n-1)$-RMGC. Since the element 2 is second in the initial permutation of the block, it follows that it is the first element in the last permutation of the block. By the construction, we now use t_n, thus making the element 2 second in the first permutation of the second block. By repeating the above arguments for each block we prove the lemma. □

Lemma 3: In any block, the last element of all the permutations is constant. The sequence of last elements in the blocks constructed is $n, n-1, \ldots, 3, 1$. The element 2 is never a last element.

Proof: The first claim is easily proved by noting that the transitions creating a block, $t_{i_1}, t_{i_2}, \ldots, t_{i_{(n-1)!-1}}$, only operate on the first $n-1$ positions of the permutations. Also, by the same logic used in the proof of the previous lemma, if the first permutation of a block is $[a_1, a_2, a_3, \ldots, a_{n-1}, a_n]$, then the last permutation in a block is $[a_2, a_1, a_3, \ldots, a_{n-1}, a_n]$, and thus the first permutation of the next block is $[a_n, a_2, a_1, a_3, \ldots, a_{n-1}]$.

It follows that if we examine the sequence containing just the *first* permutation in each block, the element a_2 remains fixed, and the rest just rotate by one position each time. By the previous lemma, the fixed element is 2, and therefore, the sequence of last elements is as claimed. □

Combining the two lemmas above, the $n-1$ blocks constructed so far form a cyclic (but not complete) n-RMGC, that we call C', which may be schematically described as as shown at the bottom of the page (where each box represents a single block, and $\rightsquigarrow$ denotes the sequence of transitions $t_{i_1}, \ldots, t_{i_{(n-1)!-1}}$).

It is now obvious that C' is not complete because it is missing exactly the $(n-1)!$ permutations containing 2 as their last element. We build a block C'' containing these permutations in the following way: we start by rotating the list of transitions $t_{i_1}, \ldots, t_{i_{(n-1)!}}$ such that its last transition is t_{n-1}.[2] For convenience, we denote the rotated sequence by $\tau_{i_1}, \ldots, \tau_{i_{(n-1)!}}$, where $\tau_{i_{(n-1)!}} = t_{n-1}$. Assume the first permutation in the block is $[a_1, a_2, \ldots, a_{n-1}, 2]$. We set the following permuta-

[2]The transition t_{n-1} must be present somewhere in the sequence or else the last element would remain constant, thus contradicting the assumption that the sequence generates a cyclic and complete $(n-1)$-RMGC.

$$\boxed{\begin{matrix}1 & 2\\ 2 & 1\\ \vdots & \vdots\\ n-1 & n-1\\ n & n\end{matrix}} \rightsquigarrow \xrightarrow[t_n]{} \boxed{\begin{matrix}n & 2\\ 2 & n\\ \vdots & \vdots\\ n-2 & n-2\\ n-1 & n-1\end{matrix}} \rightsquigarrow \xrightarrow[t_n]{} \cdots \xrightarrow[t_n]{} \boxed{\begin{matrix}3 & 2\\ 2 & 3\\ \vdots & \vdots\\ n & n\\ 1 & 1\end{matrix}}$$

tions of the block C'' to be the ones formed by the sequence of transitions $\tau_{i_1}, \ldots, \tau_{i_{(n-1)!-1}}$. Thus, the last permutation in C'' is $[a_2, \ldots, a_{n-1}, a_1, 2]$.

In C', we look for a transition of the following form: $[a_2, \ldots, a_{n-1}, 2, a_1] \rightarrow [2, a_2, \ldots, a_{n-1}, a_1]$. We contend that such a transition must surely exist: C' does not contain permutations in which 2 is last, while it does contain permutations in which 2 is next to last, and some where 2 is the first element. Since C' is cyclic, there must be at least one transition t_{n-1} pushing an element 2 from a next-to-last position to the first position. At this transition we split C' and insert C'' as follows:

$$\begin{array}{c} a_2 \\ a_3 \\ \vdots \\ a_{n-1} \\ 2 \\ a_1 \end{array} \xrightarrow[t_n]{} \boxed{\begin{array}{c} a_1 \\ a_2 \\ \vdots \\ a_{n-2} \\ a_{n-1} \\ 2 \end{array} \rightsquigarrow \begin{array}{c} a_2 \\ a_3 \\ \vdots \\ a_{n-1} \\ a_1 \\ 2 \end{array}} \xrightarrow[t_n]{} \begin{array}{c} 2 \\ a_2 \\ \vdots \\ a_{n-2} \\ a_{n-1} \\ a_1 \end{array}$$

where it is easy to see all transitions are valid. Thus, we have created C_n and to complete the recursion we have to make sure t_2 appears at least twice, but that is obvious since the sequence $t_{i_1}, \ldots, t_{i_{(n-1)!-1}}$ contains at least one occurrence of t_2, and is replicated $n-1$ times, $n \geqslant 3$. We therefore reach the following conclusion.

Theorem 4: For every integer $n \geqslant 2$ there exists a cyclic and complete n-RMGC.

Example 5: We construct a 4-RMGC by recursively using the 3-RMGC shown in Example 1, to illustrate the construction process. The sequence of transitions for the 3-RMGC in Example 1 is $t_2, t_3, t_3, t_2, t_3, t_3$. As described in the construction, in order to use this code as a basis for the 4-RMGC construction, we need to have t_2 as the last transition. We therefore rotate the sequence of transitions to be $t_3, t_3, t_2, t_3, t_3, t_2$. The resulting first three blocks, denoted C', are

$$\begin{array}{|cccccc|} \hline 1 & 3 & 2 & 3 & 1 & 2 \\ 2 & 1 & 3 & 2 & 3 & 1 \\ 3 & 2 & 1 & 1 & 2 & 3 \\ 4 & 4 & 4 & 4 & 4 & 4 \\ \hline \end{array} \quad \begin{array}{|cccccc|} \hline 4 & 1 & 2 & 1 & 4 & 2 \\ 2 & 4 & 1 & 2 & 1 & 4 \\ 1 & 2 & 4 & 4 & 2 & 1 \\ 3 & 3 & 3 & 3 & 3 & 3 \\ \hline \end{array} \quad \begin{array}{|cccccc|} \hline 3 & 4 & 2 & 4 & 3 & 2 \\ 2 & 3 & 4 & 2 & 4 & 3 \\ 4 & 2 & 3 & 3 & 2 & 4 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ \hline \end{array}$$

To create the missing fourth block, C'', the construction requires a transition sequence ending with t_3, so we use the original sequence $t_2, t_3, t_3, t_2, t_3, t_3$ shown in Example 1. To decide the starting permutation $[a_1, a_2, a_3, 2]$ of the block, we search for a transition of the form $[a_2, a_3, 2, a_1] \rightarrow [2, a_2, a_3, a_1]$ in C'. Several such transitions exist, and we arbitrarily choose $[1, 3, 2, 4] \rightarrow [2, 1, 3, 4]$ seen in the fifth and sixth columns of C'. The resulting missing block, C'', is

$$\begin{array}{|cccccc|} \hline 4 & 1 & 3 & 4 & 3 & 1 \\ 1 & 4 & 1 & 3 & 4 & 3 \\ 3 & 3 & 4 & 1 & 1 & 4 \\ 2 & 2 & 2 & 2 & 2 & 2 \\ \hline \end{array}$$

Inserting C'' between the fifth and sixth columns of C' results in the following 4-RMGC given at the bottom of the page.

III. Balanced n-RMGCs

While the construction for n-RMGCs given in the previous section is mathematically pleasing, it suffers from a practical drawback: while the $n-1$ top-charged cells are changed (having their charge level increased while going through the permutations of a single block), the bottom cell remains untouched and a large gap in charge levels develops between the least charged and most charged cells. When eventually, the least charged cell gets "pushed-to-the-top," in order to acquire the target charge level, the charging of the cell may take a long time or involve large jumps in charge level (which are prone to cause write-disturbs in neighboring cells). The balanced n-RMGC described in this section solves this problem.

A. Definition and Construction

In the current models of flash memory, it is sometimes the case that due to precision constraints in the charge placement mechanism, the actual possible charge levels are discrete. The rank-modulation scheme is not governed by such constraints, since it only needs to order cell levels unambiguously by means of comparisons, rather than compare the cell levels against predefined threshold values. However, in order to describe the following results, we will assume abstract discrete levels, that can be understood as counting the number of push-to-the-top operations executed up to the current state. In other words, each push-to-the-top increases the maximum charge level by one.

Thus, we define the function $c_i : \mathbb{N} \rightarrow \mathbb{N}$, where $c_i(p)$ is the charge level of the ith cell after the pth programming cycle. It follows that if we use transition t_j in the pth programming cycle and the ith cell is, at the time, jth from the top, then $c_i(p) > \max_k \{c_k(p-1)\}$, and for $k \neq i$, $c_k(p) = c_k(p-1)$. In an optimal setting with no overshoots, $c_i(p) = \max_k \{c_k(p-1)\} + 1$.

The *jump* in the pth round is defined as $c_i(p) - c_i(p-1)$, assuming the ith cell was the affected one. It is desirable, when programming cells, to make the jumps as small as possible. We define the *jump cost* of an n-RMGC as the maximum jump during the transitions dictated by the code. We say an n-RMGC

$$\begin{array}{cccccccccccccccccccccccc} 1 & 3 & 2 & 3 & 1 & 4 & 1 & 3 & 4 & 3 & 1 & 2 & 4 & 1 & 2 & 1 & 4 & 2 & 3 & 4 & 2 & 4 & 3 & 2 \\ 2 & 1 & 3 & 2 & 3 & 1 & 4 & 1 & 3 & 4 & 3 & 1 & 2 & 4 & 1 & 2 & 1 & 4 & 2 & 3 & 4 & 2 & 4 & 3 \\ 3 & 2 & 1 & 1 & 2 & 3 & 3 & 4 & 1 & 1 & 4 & 3 & 1 & 2 & 4 & 4 & 2 & 1 & 4 & 2 & 3 & 3 & 2 & 4 \\ 4 & 4 & 4 & 4 & 4 & 2 & 2 & 2 & 2 & 2 & 2 & 4 & 3 & 3 & 3 & 3 & 3 & 3 & 1 & 1 & 1 & 1 & 1 & 1 \end{array}$$

is *nondegenerate* if it raises each of its cells at least once. A nondegenerate n-RMGC is said to be *optimal* if its jump cost is not larger than any other nondegenerate n-RMGC.

Lemma 6: For any optimal nondegenerate n-RMGC, $n \geqslant 3$, the jump cost is at least $n+1$.

Proof: In an optimal n-RMGC, $n \geqslant 3$, we must raise the lowest cell to the top charge level at least n times. Such a jump must be at least of magnitude n. We cannot, however, do these n jumps consecutively, or else we return to the first permutation after just n steps. It follows that there must be at least one other transition t_i, $i \neq n$, and so the first t_n to be used after it jumps by at least a magnitude of $n+1$. □

We call an n-RMGC with a jump cost of $n+1$ a *balanced n-RMGC*. We now show a construction that turns any $(n-1)$-RMGC (balanced or not) into a balanced n-RMGC. The original $(n-1)$-RMGC is not required to be cyclic or complete, but if it is cyclic (complete) the resulting n-RMGC will turn out to be also cyclic (complete). The intuitive idea is to base the construction on cyclic shifts t_n that push the *bottom* to the top, and use them as often as possible. This is desirable because t_n does not introduce gaps between the charge levels, so it does not aggravate the jump cost of the cycle. Moreover, t_n partitions the set of permutations into $(n-1)!$ orbits of length n. Theorem 7 gives a construction where these orbits are traversed consecutively, based on the order given by the supporting $(n-1)$-RMGC.

Theorem 7: Given a cyclic and complete $(n-1)$-RMGC, C_{n-1}, defined by the transitions $t_{i_1}, \ldots, t_{i_{(n-1)!}}$, then the following transitions define an n-RMGC, denoted by C_n, that is cyclic, complete and *balanced*:

$$t_{j_k} = \begin{cases} t_{n-i_{\lceil k/n \rceil}+1}, & k \equiv 1 \pmod{n} \\ t_n, & \text{otherwise} \end{cases}$$

for all $k \in \{1, \ldots, n!\}$.

Proof: Let us define the abstract transition $\overrightarrow{t_i}$, $2 \leqslant i \leqslant n$, that pushes to the bottom the ith element from the bottom:

$$\overrightarrow{t_i}([a_1, \ldots, a_{n-i}, \boldsymbol{a_{n-i+1}}, a_{n-i+2}, \ldots, a_n]) = [a_1, \ldots, a_{n-i}, a_{n-i+2}, \ldots, a_n, \boldsymbol{a_{n-i+1}}].$$

Because C_{n-1} is cyclic and complete, using $\overrightarrow{t_{i_1}}, \ldots, \overrightarrow{t_{i_{(n-1)!}}}$ starting with a permutation of $[n-1]$ produces a complete cycle through all the permutations of $[n-1]$, and using them starting with a permutation of $[n]$ creates a cycle through all the $(n-1)!$ permutations of $[n]$ with the respective first element fixed, because they operate only on the last $n-1$ elements.

Because of the first element being fixed, those $(n-1)!$ permutations of $[n]$ produced by $\overrightarrow{t_{i_1}}, \ldots, \overrightarrow{t_{i_{(n-1)!}}}$, also have the property of being cyclically distinct. Thus, they are representatives of the $(n-1)!$ distinct orbits of the permutations of $[n]$ under the operation t_n, since t_n represents a simple cyclic shift when operated on a permutation of $[n]$.

Taking a permutation of $[n]$, then using the transition t_{n-i+1} once, $2 \leqslant i \leqslant n-1$, followed by $n-1$ times using t_n, is equivalent to using $\overrightarrow{t_i}$

$$\overrightarrow{t_i}(\sigma) = (\underbrace{t_n \circ \cdots \circ t_n}_{n-1 \text{ times}} \circ t_{n-i+1})(\sigma), \quad \forall \sigma \in S_n.$$

Every transition of the form t_{n-i+1}, $i \neq n$, moves us to a different orbit of t_n, while the $n-1$ consecutive executions of t_n generate all the elements of the orbit. It follows that the resulting permutations are distinct. Schematically, the construction of C_n based on C_{n-1} is

$$\underbrace{t_{n-i_1+1}, \overbrace{t_n, \ldots, t_n}^{n-1 \text{ times}}}_{\overrightarrow{t_{i_1}}}, \ldots, \underbrace{t_{n-i_{(n-1)!}+1}, \overbrace{t_n, \ldots, t_n}^{n-1 \text{ times}}}_{\overrightarrow{t_{i_{(n-1)!}}}}.$$

The code C_n is balanced, because in every block of n transitions starting with a $t_{n-i+1}, 2 \leqslant i \leqslant n-1$, we have: the transition t_{n-i+1} has a jump of $n-i+1$; the following $i-1$ transitions t_n have a jump of $n+1$, and the rest a jump of n. In addition, because C_{n-1} is cyclic and complete, it follows that C_n is also cyclic and complete. □

Theorem 8: For any $n \geqslant 2$, there exists a cyclic, complete, and balanced n-RMGC.

Proof: We can use Theorem 7 to recursively construct all the supporting n'-RMGCs, $n' \in \{n-1, \ldots, 2\}$, with the basis of the recursion being the complete cyclic 2-RMGC: $[1,2]$, $[2,1]$. □

A similar construction, but using a more involved second-order recursion, was later suggested by Etzion [9].

Example 9: Fig. 2 shows the transitions of a recursive, balanced n-RMGC for $n=4$. The permutations are represented in an $(n-1)! \times n$ matrix, where each row is an orbit generated by t_n. The transitions between rows occur when $n=4$ is the top element. Note how these permutations (the exit points of the orbits), after dropping the 4 at the top and turning them upside down, form a 3-RMGC:

$$\begin{matrix} 1 & 2 & 3 & 2 & 1 & 3 \\ 3 & 1 & 2 & 3 & 2 & 1 \\ 2 & 3 & 1 & 1 & 3 & 2 \end{matrix}.$$

This code is equivalent to the code from Example 1, up to a rotation of the transition sequence and the choice of first permutation. Fig. 3 shows the charge levels of the cells for each programming cycle, for the resulting balanced 4-RMGC.

B. Successor Function

The balanced n-RMGC can be used to implement a logic cell with $n!$ levels. This can also be understood as a counter that increments its value by one unit at a time. The function $\texttt{Successor}(n, [a_1, \ldots, a_n])$ takes as input the current permutation, and determines the transition t_i to the next permutation in the balanced recursive n-RMGC. If $n=2$, the next transition is always t_2 (line 2). Otherwise, if the top element is not n, then the current permutation is not at the exit point of its orbit, therefore the next transition is t_n (line 5). However, if n is the top element, then the transition is defined by the supporting cycle. The function is called recursively, on the reflected permutation of $[a_2, \ldots, a_n]$ (line 7).

An important practical aspect is the average number of steps required to decide which transition generates the next permutation from the current one. A *step* is defined as a single query

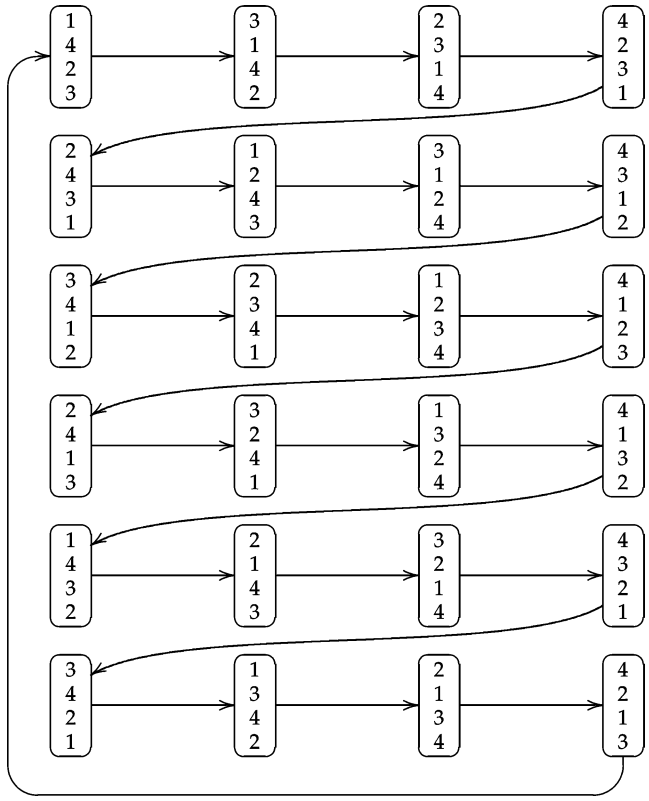

Fig. 2. Balanced 4-RMGC.

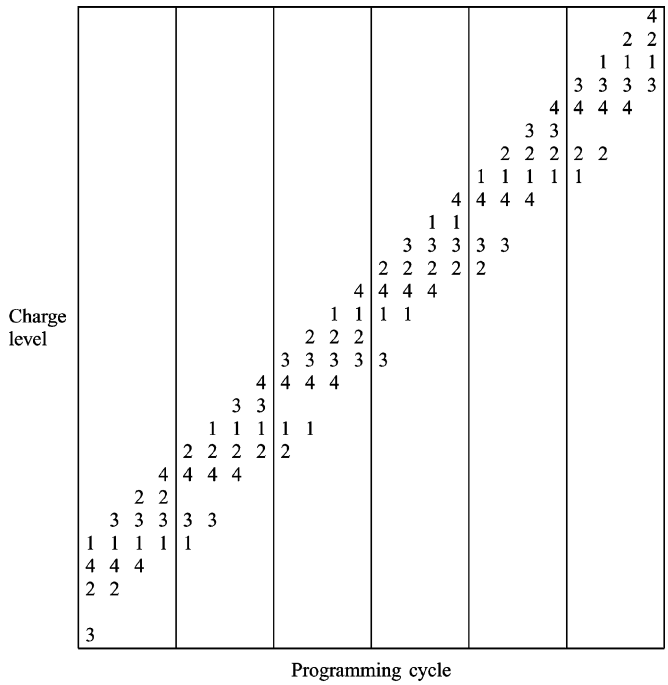

Fig. 3. Charge level growth for the balanced 4-RMGC.

of the form "what is the ith highest charged cell?" namely the comparison in line 4.

The function `Successor` is asymptotically optimal with respect to this measure:

Function `Successor`$(n, [a_1, \ldots, a_n])$

input : $n \in \mathbb{N}, n \geqslant 2$; permutation $[a_1, \ldots, a_n]$
output : $i \in \{2, \ldots, n\}$ that determines the transition t_i to the next permutation in the balanced recursive n-RMGC

```
1 if n = 2 then
2 |  return 2
3 else
4 |  if a_1 ≠ n then
5 |  |  return n
6 |  else
7 |  |  return n − Successor(n − 1, [a_n, a_{n−1}, ..., a_2]) + 1
```

Theorem 10: In the function `Successor`, the asymptotic average number of steps to create the successor of a given permutation is one.

Proof: A fraction of $\frac{n-1}{n}$ of the transitions are t_n, and these occur whenever the cell n is not the highest charged one, and they are determined in just one step. Of the cases where n is highest charged, by recursion, a fraction $\frac{n-2}{n-1}$ of the transitions are determined by just one more step, and so on. At the basis of the recursion, permutations over two elements require zero steps. Equivalently, the query "is a_1 equal to n" is performed for every permutation, therefore $n!$ times; the query "is a_n equal to $n-1$" is performed only for $\frac{1}{n}n!$ permutations, therefore $(n-1)!$ times, and so on. Thus, the total number of queries is $\sum_{i=3}^{n} i!$. Since $\lim_{n\to\infty} \sum_{i=3}^{n} i!/n! = 1$, the asymptotic average number of steps to generate the next permutation is as stated. □

C. Ranking Permutations

In order to complete the design of the logic cell, we need to define the correspondence between a permutation and its rank in the balanced n-RMGC. This problem is similar to that of ranking permutations in lexicographic order. We will first review the factoradic numbering system, and then present a new numbering system that we call *b-factoradic*, induced by the balanced n-RMGC construction.

1) Review of the Factoradic Numbering System: The factoradic is a mixed radix numbering system. The earliest reference appears in [26]. Lehmer [27] describes algorithms that make the correspondence between permutations and factoradic.

Any integer number $m \in \{0, 1, \ldots, n! - 1\}$ can be represented in the factoradic system by the digits $d_{n-1}d_{n-2}\ldots d_1 d_0$, where $d_i \in \{0, \ldots, i\}$ for $i \in \{0, \ldots, n-1\}$, and the weight of d_i is $i!$ (with the convention that $0! = 1$). The digit d_0 is always 0, and is sometimes omitted

$$m = \sum_{i=0}^{n-1} d_i i!.$$

Any permutation $[a_1, \ldots, a_n]$ has a unique factoradic representation that gives its position in the lexicographic ordering. The digits d_i are in this case the number of elements smaller than a_{n-i} that are to the right of a_{n-i}. They are therefore inversion counts, and the factoradic representation is an inversion table (or vector) [15].

There is a large literature devoted to the study of ranking permutations from a complexity perspective. Translating between factoradic and decimal representation can be done in

$O(n)$ arithmetic operations. The bottleneck is how to translate efficiently between permutations and factoradic. A naive approach similar to the simple algorithms described in [27] requires $O(n^2)$. This can be improved to $O(n \log n)$ by using merge–sort counting, or a binary search tree, or modular arithmetic, all techniques described in [25]. This can be further improved to $O(n \log n / \log \log n)$[30], by using the special data structure of Dietz [7]. In [30] linear time complexity is also achieved by departing from lexicographic ordering. A linear time complexity is finally achieved in [28], by using the fact that the word size has to be $O(n \log n)$ in order to represent numbers up to $n!$, and by hiding rich data structures in integers of this size.

2) B-Factoradic—A New Numbering System: We will now describe how to index permutations of the balanced recursive n-RMGC with numbers from $\{0, \ldots, n! - 1\}$, such that consecutive permutations in the cycle have consecutive ranks modulo $n!$. The permutation that gets index 0 is a special permutation that starts a new orbit generated by t_n, and also starts a new orbit in any of the recursive supporting n'-RMGCs, $n' \in \{n-1, \ldots, 2\}$.

The rank of a permutation is determined by its position in the orbit of t_n, and by the rank of the orbit, as given by the rank of the supporting permutation of $[n-1]$. The position of a permutation inside an orbit of t_n is given by the position of n. If the current permutation is $[a_1, \ldots, a_n]$ and $a_i = n$ for $i \in \{1, \ldots, n\}$, then the position in the current orbit of t_n is $(i-2) \bmod n$ (because the orbit starts with n in position a_2). The index of the current orbit is given by the rank of the supporting permutation of $[n-1]$, namely, the rank of $[a_{i-1}, a_{i-2} \ldots, a_1, a_n, a_{n-1}, \ldots, a_{i+1}]$ (notice that the permutation of $[n-1]$ is reflected). Therefore, if $a_i = n$, then

$$\begin{aligned} &\texttt{Rank}([a_1, \ldots, a_n]) \\ &\quad = ((i-2) \bmod n) \\ &\qquad + n \cdot \texttt{Rank}([a_{i-1}, \ldots, a_1, a_n, \ldots, a_{i+1}]). \end{aligned} \tag{1}$$

The above formula can be used recursively to determine the rank of the permutations from the supporting balanced n'-RMGCs, for $n' \in \{n-1, \ldots, 2\}$. It now becomes clear what permutation should take rank 0. The highest element in every supporting RMGC should be in the second position, therefore, $a_2 = n, a_n = n-1, a_3 = n-2, a_{n-1} = n-3$, and so on, and $a_1 = 1$. Therefore, $[1, n, n-2, n-4, \ldots, n-3, n-1]$ gets the rank 0. See Example 9 for the construction of the recursive and balanced 4-RMGC where the permutation $[1, 4, 2, 3]$ has rank 0. Equation (1) induces a new numbering system that we call *b-factoradic* (backwards factoradic). A number $m \in \{0, \ldots, n! - 1\}$ can be represented by the digits $b_{n-1} b_{n-2} \ldots b_1 b_0$, where $b_i \in \{0, \ldots, n-1-i\}$ and the weight of b_i is $n!/(n-i)!$. In this case b_{n-1} is always 0 and can be omitted. It is easy to verify that this is a valid numbering system, therefore, any $m \in \{0, \ldots, n! - 1\}$ has a unique b-factoradic representation such that

$$m = \sum_{i=0}^{n-1} b_i \frac{n!}{(n-i)!}.$$

The weights of the b-factoradic are sometimes called "falling factorials," and can be represented succinctly by the Pochhammer symbol.

Example 11: Let $n = 6$ and $\sigma = [2, 5, 4, 3, 6, 1]$ be the current permutation. We can find its b-factoradic representation $b_5 b_4 b_3 b_2 b_1 b_0$ as follows. We start from the least significant digit b_0, which is given by the position of 6 minus 2 modulo 6, so $b_0 = (5-2) \bmod 6 = 3$ (here we keep the elements of the permutation indexed from 1 to n). We now recurse on the residual permutation of five elements, $\sigma_5 = [3, 4, 5, 2, 1]$ (notice the reflected reading of this permutation, from 6 towards the left). Now b_1 is given by the position of 5; $b_1 = (3-2) \bmod 5 = 1$. The residual permutation is $\sigma_4 = [4, 3, 1, 2]$, therefore, $b_2 = (1-2) \bmod 4 = 3$. For the next step, $\sigma_3 = [2, 1, 3]$ and $b_3 = (3-2) \bmod 3 = 1$. Finally, $\sigma_2 = [1, 2]$ and $b_4 = (2-2) \bmod 2 = 0$. As always $b_{n-1} = b_5 = 0$. The b-factoradic representation is therefore $0_5\ 0_4\ 1_3\ 3_2\ 1_1\ 3_0$, where the subscript indicates the position of the digit. Going from a b-factoradic representation to a permutation of the balanced n-RMGC can follow a similar reversed procedure.

The procedure of Example 11 can be formalized algorithmically, however, its time complexity is $O(n^2)$, similar to the naive algorithms specific to translations between permutations in lexicographic order and factoradic. We can in principle use all the available results for factoradic, described previously, to achieve time complexity of $O(n \log n)$ or lower. However, we are not going to repeat all those methods here, but rather describe a linear time procedure that takes a permutation and its factoradic as input and outputs the b-factoradic. We can thus leverage directly all the results available for factoradic, and use them to determine the current symbol of a logic cell.

The procedure `Factoradic-To-BFactoradic` exploits the fact that the inversion counts are already given by the factoradic representation, and they can be used to compute directly the digits of the b-factoradic. A b-factoradic digit b_k is a count of the elements smaller than $n-k$ that lie between $n-k+1$ and $n-k$ when the permutation is viewed as a cycle. The direction of the count alternates for even and odd values of k. The inverse of the input permutation σ can be computed in $O(n)$ time (line 1). The position of every element of the permutation can then be computed in constant time (lines 2 and 5). The test in line 6 decides if we count towards the right or left starting from the position i that holds element $n-k+1$, until we reach position j that holds element $n-k$. By working out the cases when $i < j$ and $j < i$ we obtain the formulas in lines 7 and 9. Since this computation takes a constant number of arithmetic operations, the entire algorithm takes $O(n)$ time.

Unranking, namely, going from a number in $\{0, \ldots, n! - 1\}$ to a permutation in balanced order is likely to never be necessary in practice, since the logic cell is designed to be a counter. However, for completeness, we describe the simplest $O(n^2)$ procedure `Unrank`, that takes a b-factoradic as input and produces the corresponding permutation. The procedure uses variable p to simulate the cyclic counting of elements smaller than the current one. The direction of the counting alternates, based on the test in line 4.

Procedure Factoradic-To-BFactoradic

input : $n \in \mathbb{N}, n \geqslant 2$; permutation $\sigma = [a_1, \ldots, a_n]$ and its factoradic $d_{n-1} \ldots d_1 d_0$.
output : b-factoradic $b_{n-1} \ldots b_1 b_0$ corresponding to the index of σ in the balanced recursive n-RMGC starting with $[1, n, n-2, \ldots, n-3, n-1]$.

compute the inverse permutation σ^{-1}
$i \leftarrow \sigma^{-1}(n)$ // namely $a_i = n$
$b_0 \leftarrow (n - d_{n-i} - 2) \mod n$
for $k \leftarrow 1$ **to** $n-2$ **do**
  $j \leftarrow \sigma^{-1}(n-k)$ // namely $a_j = n-k$
  if k *is even* **then**
    $b_k \leftarrow (d_{n-i} - d_{n-j} - 2) \mod (n-k)$
  else
    $b_k \leftarrow (d_{n-j} - d_{n-i} - 1) \mod (n-k)$
  $i \leftarrow j$
$b_{n-1} \leftarrow 0$

Procedure Unrank

input : $n \in \mathbb{N}, n \geqslant 2$, b-factoradic $b_{n-1}, \ldots, b_0$
output : Permutation $[a_1, \ldots, a_n]$

Initialize $[a_1, \ldots, a_n] \leftarrow [1, \ldots, 1]$
$p \leftarrow 0$
for $i \leftarrow n$ **down to** 2 **do**
  if $i \equiv n \mod 1$ **then**
    for $j \leftarrow 1$ **to** $b_{n-i} + 2$ **do**
      $p \leftarrow (p+1) \mod n$
      while $a_p \neq 1$ **do**
        $p \leftarrow (p+1) \mod n$
  else
    for $j \leftarrow 1$ **to** $b_{n-i} + 2$ **do**
      $p \leftarrow (p-1) \mod n$
      while $a_p \neq 1$ **do**
        $p \leftarrow (p-1) \mod n$
  $a_p \leftarrow i$

IV. Rewriting With Rank-Modulation Codes

In Gray codes, the states transit along a well-designed path. What if we want to use the rank-modulation scheme to store data, and allow the data to be modified in arbitrary ways? Consider an information symbol $i \in [\ell]$ that is stored using n cells. In general, ℓ might be smaller than $n!$, so we might end up having permutations that are not used. On the other hand, we can map several distinct permutations to the same symbol i in order to reduce the rewrite cost. We let $W_n \subseteq S_n$ denote the set of states (i.e., the set of permutations) that are used to represent information symbols. We define two functions, an *interpretation function*, φ, and an *update function*, μ.

Definition 12: The interpretation function $\varphi : W_n \rightarrow [\ell]$ maps every state $s \in W_n$ to a value $\varphi(s)$ in $[\ell]$. Given an "old state" $s \in W_n$ and a "new information symbol" $i \in [\ell]$, the update function $\mu : W_n \times [\ell] \rightarrow W_n$ produces a state $\mu(s,i)$ such that $\varphi(\mu(s,i)) = i$.

When we use n cells to store an information symbol, the permutation induced by the charge levels of the cells represents the information through the interpretation function. We can start the process by programming some arbitrary initial permutation in the flash cells. Whenever we want to change the stored information symbol, the permutation is changed using the "push-to-the-top" operations based on the update function. We can keep changing the stored information as long as we do not reach the maximal charge level possible in any of the cells. Therefore, the number of "push-to-the-top" operations in each rewrite operation determines not only the rewriting delay but also how much closer the highest cell-charge level is to the system limit (and therefore how much closer the cell block is to the next costly erase operation). Thus, the objective of the coding scheme is to minimize the number of "push-to-the-top" operations.

Definition 13: Given two states $s_1, s_2 \in W_n$, the *cost* of changing s_1 into s_2, denoted $\alpha(s_1 \rightarrow s_2)$, is defined as the minimum number of "push-to-the-top" operations needed to change s_1 into s_2.

For example, $\alpha([1,2,3] \rightarrow [2,1,3]) = 1$, $\alpha([1,2,3] \rightarrow [3,2,1]) = 2$. We define two important measures: the worst case rewrite cost and the average rewrite cost.

Definition 14: The worst case rewrite cost is defined as $\max_{s \in W_n, i \in [\ell]} \alpha(s \rightarrow \mu(s,i))$. Assume input symbols are independent and identically distributed (i.i.d.) random variables having value $i \in [\ell]$ with probability p_i. Given a fixed $s \in W_n$, the average rewrite cost given s is defined as $A(s) = \sum_{i \in [\ell]} p_i \alpha(s \rightarrow \mu(s,i))$. If we further assume some stationary probability distribution over the states W_n, where we denote the probability of state s as q_s, then the average rewrite cost of the code is defined as $\sum_{s \in W_n} q_s A(s)$. (Note that for all $i \in [\ell]$, $p_i = \sum_{s \in \{s' \in W_n \mid \varphi(s') = i\}} q_s$.)

In this section, we present a code that minimizes the worst case rewrite cost. In Section IV-A, we focus on codes with good average rewrite cost.

A. Lower Bound

We start by presenting a lower bound on the worst case rewrite cost. Define the *transition graph* $G = (V, E)$ as a directed graph with $V = S_n$, that is, with $n!$ vertices representing the permutations in S_n. For any $u, v \in V$, there is a directed edge from u to v iff $\alpha(u \rightarrow v) = 1$. G is a regular digraph, because every vertex has $n-1$ incoming edges and $n-1$ outgoing edges. The diameter of G is $\max_{u,v \in V} \alpha(u \rightarrow v) = n-1$.

Given a vertex $u \in V$ and an integer $r \in \{0, 1, \ldots, n-1\}$, define the *ball* centered at u with radius r as $\mathcal{B}_r^n(u) = \{v \in V \mid \alpha(u \rightarrow v) \leqslant r\}$, and define the *sphere* centered at u with radius r as $\mathcal{S}_r^n(u) = \{v \in V \mid \alpha(u \rightarrow v) = r\}$. Clearly

$$\mathcal{B}_r^n(u) = \bigcup_{0 \leqslant i \leqslant r} \mathcal{S}_r^n(u).$$

By a simple relabeling argument, both $|\mathcal{B}_r^n(u)|$ and $|\mathcal{S}_r^n(u)|$ are independent of u, and so will be denoted by $|\mathcal{B}_r^n|$ and $|\mathcal{S}_r^n|$ respectively.

Lemma 15: For any $0 \leqslant r \leqslant n-1$

$$|\mathcal{B}_r^n| = \frac{n!}{(n-r)!}$$

$$|\mathcal{S}_r^n| = \begin{cases} 1, & r = 0 \\ \frac{n!}{(n-r)!} - \frac{n!}{(n-r+1)!}, & 1 \leqslant r \leqslant n-1. \end{cases}$$

Proof: Fix a permutation $u \in V$. Let P_u be the set of permutations having the following property: for each permutation $v \in P_u$, the elements appearing in its last $n-r$ positions appear

in the same relative order in u. For example, if $n = 5$, $r = 2$, $u = [1, 2, 3, 4, 5]$, and $v = [5, 2, 1, 3, 4]$, the last three elements of v—namely, $1, 3, 4$—have the same relative order in u. It is easy to see that given u, when the elements occupying the first r positions in $v \in P_u$ are chosen, the last $n-r$ positions become fixed. There are $n(n-1)\cdots(n-r+1)$ choices for occupying the first r positions of $v \in P_u$, hence, $|P_u| = \frac{n!}{(n-r)!}$. We will show that a vertex v is in $\mathcal{B}_r^n(u)$ if and only if $v \in P_u$.

Suppose $v \in \mathcal{B}_r^n(u)$. It follows that v can be obtained from u with at most r "push-to-the-top" operations. Those elements pushed to the top appear in the first r positions of v, so the last $n - r$ positions of v contain elements which have the same relative order in u, thus, $v \in P_u$.

Now suppose $v \in P_u$. For $i \in [n]$, let v_i denote the element in the ith position of v. One can transform u into v by sequentially pushing $v_r, v_{r-1}, \ldots, v_1$ to the top. Hence, $v \in \mathcal{B}_r^n(u)$.

We conclude that $|\mathcal{B}_r^n(u)| = |P| = \frac{n!}{(n-r)!}$. Since $\mathcal{B}_r^n(u) = \bigcup_{0 \leqslant i \leqslant r} \mathcal{S}_i^n(u)$, the second claim follows. □

The following lemma presents a lower bound on the worst case rewrite cost.

Lemma 16: Fix integers n and ℓ, and define $\rho(n, \ell)$ to be the smallest integer such that $\left|\mathcal{B}_{\rho(n,\ell)}^n\right| \geqslant \ell$. For any code W_n and any state $s \in W_n$, there exists $i \in [\ell]$ such that $\alpha(s \to \mu(s, i)) \geqslant \rho(n, \ell)$, i.e., the worst case rewrite cost of any code is at least $\rho(n, \ell)$.

Proof: By the definition of $\rho(n, \ell)$, $|\mathcal{B}_{\rho(n,\ell)-1}^n| < \ell$. Hence, we can choose $i \in [\ell] \setminus \{\varphi(s') \mid s' \in \mathcal{B}_{\rho(n,\ell)-1}^n(s)\}$. Clearly, by our choice $\alpha(s \to \mu(s, i)) \geqslant \rho(n, \ell)$. □

B. Optimal Code

We now present a code construction. It will be shown that the code has optimal worst case performance. First, let us define the following notation.

Definition 17: A *prefix sequence* $a = [a^{(1)}, a^{(2)}, \ldots, a^{(m)}]$ is a sequence of $m \leqslant n$ distinct symbols from $[n]$. The *prefix set* $P_n(a) \subseteq S_n$ is defined as all the permutations in S_n which start with the sequence a.

We are now in a position to construct the code.

Construction 18: Arbitrarily choose ℓ distinct prefix sequences, $a_1, \ldots, a_\ell$, each of length $\rho(n, \ell)$. Let us define $W_n = \bigcup_{i \in [\ell]} P_n(a_i)$ and map the states of $P_n(a_i)$ to i, i.e., for each $i \in [\ell]$ and $s \in P_n(a_i)$, set $\varphi(s) = i$.

Finally, to construct the update function μ, given $s \in W_n$ and some $i \in [\ell]$, we do the following: let $[a_i^{(1)}, a_i^{(2)}, \ldots, a_i^{(\rho(n,\ell))}]$ be the first $\rho(n, \ell)$ elements which appear in all the permutations in $P_n(a_i)$. Apply push-to-the-top on the elements $a_i^{(\rho(n,\ell))}, \ldots, a_i^{(2)}, a_i^{(1)}$ in s to get a permutation $s' \in P_n(a_i)$ for which, clearly, $\varphi(s') = i$. Set $\mu(s, i) = s'$.

Theorem 19: The code in Construction 18 is optimal in terms of minimizing the worst case rewrite cost.

Proof: First, the number of length m prefix sequences is $\frac{n!}{(n-m)!} = |\mathcal{B}_m^n|$. By definition, the number of prefix sequences of length $\rho(n, \ell)$ is at least ℓ, which allows the first of the construction. To complete the proof, it is obvious from the description of μ that the worst case rewrite cost of the construction is at most $\rho(n, \ell)$. By Lemma 16 this is also the best we can hope for. □

Example 20: Let $n = 3$, $\ell = 3$. Since $\left|\mathcal{B}_1^3\right| = 3$, it follows that $\rho(n, \ell) = 1$. We partition the $n! = 6$ states into $\frac{n!}{(n-\rho(n,\ell))!} = 3$ sets, which induce the mapping

$$P_3([1]) = \{[1, 2, 3], [1, 3, 2]\} \mapsto 1$$
$$P_3([2]) = \{[2, 1, 3], [2, 3, 1]\} \mapsto 2$$
$$P_3([3]) = \{[3, 1, 2], [3, 2, 1]\} \mapsto 3.$$

The cost of any rewrite operation is at most 1.

V. Optimizing Average Rewrite Cost

In this section, we study codes that minimize the average rewrite cost. We first present a prefix-free code that is optimal in terms of its own design objective. Then, we show that this prefix-free code minimizes the average rewrite cost with an approximation ratio 3 if $\ell \leqslant n!/2$, and when $\ell \leqslant n!/6$, the approximation ratio is further reduced to 2.

A. Prefix-Free Code

The prefix-free code we propose consists of ℓ prefix sets $P_n(a_1), \ldots, P_n(a_\ell)$ (induced by ℓ prefix sequences $a_1, \ldots, a_\ell$) which we will map to the ℓ input symbols: for every $i \in [\ell]$ and $s \in P_n(a_i)$, we set $\varphi(s) = i$. Unlike in the previous section, the prefix sequences are no longer necessarily of the same length. We do, however, require that no prefix sequence be the prefix of another.

A prefix-free code can be represented by a tree. First, let us define a full permutation tree T as follows. The vertices in T are placed in $n + 1$ layers, where the root is in layer 0 and the leaves are in layer n. Edges only exist between adjacent layers. For $i = 0, 1, \ldots, n-1$, a vertex in layer i has $n-i$ children. The edges are labeled in such a way that every leaf corresponds to a permutation from S_n which may be constructed from the labels on the edges from the root to the leaf. An example is given in Fig. 4(a). A prefix-free code corresponds to a subtree C of T (see Fig. 4(b) for an example). Every leaf is mapped to a prefix sequence which equals the string of labels as read on the path from the root to the leaf.

For $i \in [\ell]$, let a_i denote the prefix sequence representing i, and let $|a_i|$ denote its length. For example, the prefix sequences in Fig. 4(b) have minimum length 1 and maximum length 3. The average codeword length is defined as

$$\sum_{i=1}^{\ell} p_i \, |a_i| \, .$$

Here, the probabilities p_i are as defined before, that is, information symbols are i.i.d. random variables having value $i \in [\ell]$ with probability p_i. We can see that with the prefix-free code, for every rewrite operation (namely, regardless of the old permutation before the rewriting), the expected rewrite cost is upper-bounded by $\sum_{i=1}^{\ell} p_i \, |a_i|$. Our objective is to design a prefix-free code that minimizes its average codeword length.

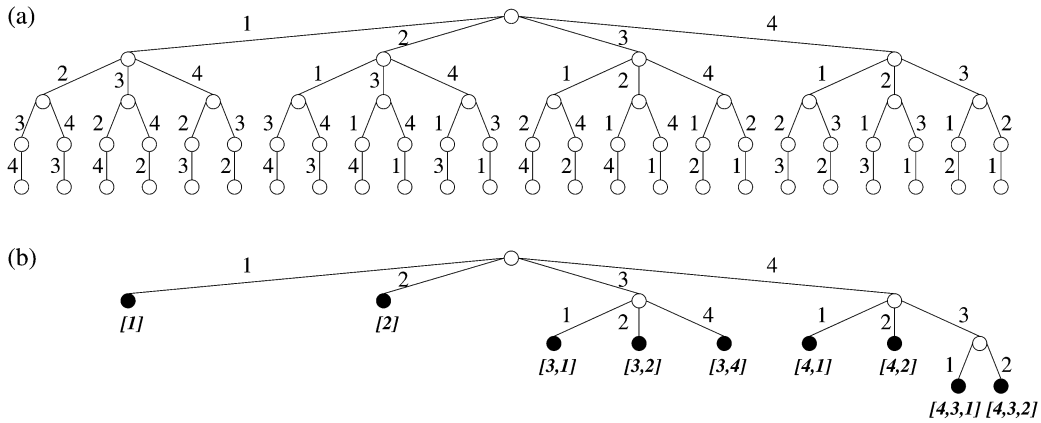

Fig. 4. Prefix-free rank-modulation code for $n = 4$ and $\ell = 9$. (a) The full permutation tree T. (b) A prefix-free code represented by a subtree C of T. The leaves represent the prefix sequences, which are displayed beside the leaves.

Example 21: Let $n = 4$ and $\ell = 9$, and let the prefix-free code be as shown in Fig. 4(b). We can map the information symbols $i \in [\ell]$ to the prefix sequences a_i as follows:

$$\begin{array}{lll} a_1 = [1] & a_2 = [2] & a_3 = [3,1] \\ a_4 = [3,2] & a_5 = [3,4] & a_6 = [4,1] \\ a_7 = [4,2] & a_8 = [4,3,1] & a_9 = [4,3,2]. \end{array}$$

Then, the mapping from the permutations to the information symbols is

$$\begin{aligned} P_4([1]) = \{[1,2,3,4],[1,2,4,3],\ldots,[1,4,3,2]\} &\mapsto 1 \\ P_4([2]) = \{[2,1,3,4],[2,1,4,3],\ldots,[2,4,3,1]\} &\mapsto 2 \\ P_4([3,1]) = \{[3,1,2,4],[3,1,4,2]\} &\mapsto 3 \\ P_4([3,2]) = \{[3,2,1,4],[3,2,4,1]\} &\mapsto 4 \\ P_4([3,4]) = \{[3,4,1,2],[3,4,2,1]\} &\mapsto 5 \\ P_4([4,1]) = \{[4,1,2,3],[4,1,3,2]\} &\mapsto 6 \\ P_4([4,2]) = \{[4,2,1,3],[4,2,3,1]\} &\mapsto 7 \\ P_4([4,3,1]) = \{[4,3,1,2]\} &\mapsto 8 \\ P_4([4,3,2]) = \{[4,3,2,1]\} &\mapsto 9. \end{aligned}$$

Assume that the current state of the cells is $[1,2,3,4] \in P_4([1])$, representing the information symbol 1. If we want to rewrite the information symbol as 8, we can shift cells 3, 4 to the top to change the state to $[4,3,1,2] \in P_4([4,3,1])$. This rewrite cost is 2, which does not exceed $|a_8| = 3$. In general, given any current state, considering all the possible rewrites, the expected rewrite cost is always less than $\sum_{i=1}^{\ell} p_i |a_i|$, the average codeword length.

The optimal prefix-free code cannot be constructed with a greedy algorithm like the Huffman code [19], because the internal nodes in different layers of the full permutation tree T have different degrees, making the distribution of the vertex degrees in the code tree C initially unknown. The Huffman code is a well-known variable-length prefix-free code, and many variations of it have been studied. In [20], the Huffman code construction was generalized, assuming that the vertex-degree distribution in the code tree is given. In [1], prefix-free codes for infinite alphabets and nonlinear costs were presented. When the letters of the encoding alphabet have unequal lengths, only exponential-time algorithms are known, and it is not known yet whether this problem is NP-hard [12]. To construct prefix-free codes for our problem, which minimize the average codeword length, we present a dynamic-programming algorithm of time complexity $O(n\ell^4)$. Note that without loss of generality, we can assume the length of any prefix sequence to be at most $n-1$.

The algorithm computes a set of functions $\mathsf{opt}_i(x,m)$, for $i = 1,2,\ldots,n-1$, $x = 0,1,\ldots,\ell$, and $m = 0,1,\ldots,\min\{\ell, n!/(n-i)!\}$. We interpret the meaning of $\mathsf{opt}_i(x,m)$ as follows. We take a subtree of T that contains the root. The subtree has exactly x leaves in the layers $i, i+1,\ldots,n-1$. It also has at most m vertices in the layer i. We let the x leaves represent the x letters from the alphabet $[\ell]$ with the lowest probabilities p_j: the further the leaf is from the root, the lower the corresponding probability is. Those leaves also form x prefix sequences, and we call their weighted average length (where the probabilities p_j are weights) the *value* of the subtree. The minimum value of such a subtree (among all such subtrees) is defined to be $\mathsf{opt}_i(x,m)$. In other words, $\mathsf{opt}_i(x,m)$ is the minimum average prefix-sequence length when we assign a subset of prefix sequences to a subtree of T (in the way described above). Clearly, the minimum average codeword length of a prefix-free code equals $\mathsf{opt}_1(\ell,n)$.

Without loss of generality, let us assume that $p_1 \leqslant p_2 \leqslant \cdots \leqslant p_\ell$. It can be seen that the following recursions hold.

- When $i = n-1$ and $m \geqslant x > 0$
$$\mathsf{opt}_i(x,m) = (n-1)\sum_{k=1}^{x} p_k.$$
- When $i \geqslant 1$ and $x = 0$
$$\mathsf{opt}_i(x,m) = 0.$$
- When $x > m \cdot (n-i)!$
$$\mathsf{opt}_i(x,m) = \infty.$$
- When $i < n-1$ and $0 < x \leqslant m \cdot (n-i)!$

$$\mathsf{opt}_i(x,m) = \min_{0 \leqslant j \leqslant \min\{x,m\}} \Big\{ \mathsf{opt}_{i+1}(x-j,$$

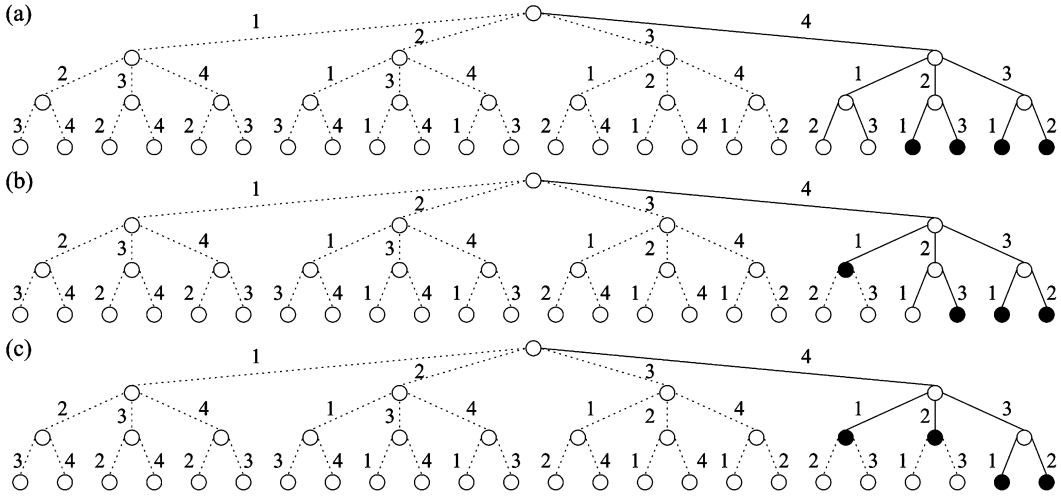

Fig. 5. Three cases for computing $\mathsf{opt}_2(4,3)$ in Example 22. The solid-line edges are in the subtree. The dotted-line edges are the remaining edges in the full-permutation tree T. The leaves in the subtree are shown as black vertices. (a) No leaf in layer 2. (b) One leaf in layer 2. (c) Two leaves in layer 2.

$$\min\{\ell,(m-j)\cdot(n-i)\})+\sum_{k=x-j+1}^{x} ip_k\Bigg\}.$$

The last recursion holds because a subtree with x leaves in layers $i, i+1, \ldots, n-1$ and at most m vertices in layer i can have $0, 1, \ldots, \min\{x, m\}$ leaves in layer i.

The algorithm first computes $\mathsf{opt}_{n-1}(x,m)$, then $\mathsf{opt}_{n-2}(x,m)$, and so on, until it finally computes $\mathsf{opt}_1(\ell,n)$, by using the above recursions. Given these values, it is straightforward to determine in the optimal code, how many prefix sequences are in each layer, and therefore determine the optimal code itself. It is easy to see that the algorithm returns an optimal code in time $O(n\ell^4)$.

Example 22: Let $n=4$ and $\ell=9$, and let us assume that $p_1 \leqslant p_2 \leqslant \cdots \leqslant p_\ell$. As an example, let us consider how to compute $\mathsf{opt}_2(4,3)$.

By definition, $\mathsf{opt}_2(4,3)$ corresponds to a subtree of T with a total of four leaves in layer 2 and layer 3, and with at most three vertices in layer 2. Thus, there are four cases to consider: either there are zero, one, two, or three leaves in layer 2. The corresponding subtrees in the first three cases are as shown in Fig. 5(a)–(c), respectively. The fourth case is actually impossible, because it leaves no place for the fourth leaf to exist in the subtree.

If layer 2 has i leaves $(0 \leqslant i \leqslant 3)$, then layer 3 has $4-i$ leaves and there can be at most $(3-i)\cdot 2$ vertices in layer 3 of the subtree. To assign p_1, p_2, p_3, p_4 to the four leaves and minimize the weighted average distance of the leaves to the root (which is defined as $\mathsf{opt}_2(4,3)$), among the four cases mentioned above, we choose the case that minimizes that weighted average distance. Therefore

$$\begin{aligned}
&\mathsf{opt}_2(4,3)\\
&\quad=\min\{\mathsf{opt}_3(4-0,(3-0)\cdot 2),\\
&\qquad\mathsf{opt}_3(4-1,(3-1)\cdot 2)+2p_4,\\
&\qquad\mathsf{opt}_3(4-2,(3-2)\cdot 2)+2p_3+2p_4,\\
&\qquad\mathsf{opt}_3(4-3,(3-3)\cdot 2)+2p_2+2p_3+2p_4\}\\
&\quad=\min\{3p_1+3p_2+3p_3+3p_4,\\
&\qquad 3p_1+3p_2+3p_3+2p_4,\\
&\qquad 3p_1+3p_2+2p_3+2p_4,\\
&\qquad \infty\}\\
&\quad=3p_1+3p_2+2p_3+2p_4.
\end{aligned}$$

Now assume that after computing all the $\mathsf{opt}_i(x,m)$'s, we find that

$$\mathsf{opt}_1(9,4)=\mathsf{opt}_2(9-2,(4-2)\cdot 3)+(p_8+p_9).$$

That means that in the optimal code tree, there are two leaves in layer 1. If we further assume that

$$\begin{aligned}
\mathsf{opt}_2(7,6)=\mathsf{opt}_3(7-5,(6-5)\cdot 2)\\
+(2p_3+2p_4+2p_5+2p_6+2p_7)
\end{aligned}$$

we can determine that there are five leaves in layer 2, and the optimal code tree will be as shown in Fig. 4(b).

We can use the prefix-free code for rewriting in the following way: to change the information symbol to $i \in [\ell]$, push at most $|a_i|$ cells to the top so that the $|a_i|$ top-ranked cells are the same as the codeword a_i.

B. Performance Analysis

We now analyze the average rewrite cost of the prefix-free code. We obviously have $\ell \leqslant n!$. When $\ell = n!$, the code design becomes trivial—each permutation is assigned a distinct input symbol. In this subsection, we prove that the prefix-free code has good approximation ratios under mild conditions: when $\ell \leqslant n!/2$, the average rewrite cost of a prefix-free code (that was built to minimize its average codeword length) is at most three times the average rewrite cost of an *optimal code* (i.e., a code that minimizes the average rewrite cost), and when $\ell \leqslant n!/6$, the approximation ratio is further reduced to 2.

Loosely speaking, our strategy for proving this approximation ratio involves an initial simple bound on the rewrite cost of *any* code when considering a rewrite operation starting with a

stored symbol $i \in [\ell]$. We then proceed to define a *prefix-free code* which locally optimizes (up to the approximation ratio) rewrite operations starting with stored symbol i. Finally, we introduce the globally optimal prefix-free code of the previous section, which optimizes the average rewrite cost, and show that it is still within the correct approximation ratio.

We start by bounding from below the average rewrite cost of any code, depending on the currently stored information symbol. Suppose we are using some code $\mathcal{C}$ with an interpretation function $\varphi^{\mathcal{C}}$ and an update function $\mu^{\mathcal{C}}$. Furthermore, let us assume the currently stored information symbol is $i \in [\ell]$ in some state $s_i \in S_n$, i.e., $\varphi^{\mathcal{C}}(s_i) = i$. We want to consider rewrite operations which are meant to store the value $j \in [\ell]$ instead of i, for all $j \neq i$. Without loss of generality, assume that the probabilities of information symbols are monotonically decreasing

$$p_1 \geqslant p_2 \geqslant \cdots \geqslant p_\ell.$$

Let us denote by $s'_1, \ldots, s'_{\ell-1} \in S_n \setminus \{s_i\}$ the $\ell - 1$ closest permutations to s_i ordered by increasing distance, i.e.,

$$\alpha(s_i \to s'_1) \leqslant \alpha(s_i \to s'_2) \leqslant \cdots \leqslant \alpha(s_i \to s'_{\ell-1})$$

and denote $\delta_j = \alpha(s_i \to s'_j)$ for every $j \in [\ell - 1]$. We note that $\delta_1, \ldots, \delta_{\ell-1}$ are independent of the choice of s_i, and furthermore, that $\delta_1 = 1$ while $\delta_{\ell-1} = \rho(n, \ell)$.

The average rewrite cost of a stored symbol $i \in [\ell]$ using a code $\mathcal{C}$ is the weighted sum

$$\xi^{\mathcal{C}}(i) = \sum_{j \in [\ell], j \neq i} p_j \alpha\left(s_i \to \mu^{\mathcal{C}}(s_i, j)\right).$$

This sum is minimized when $\left\{\mu^{\mathcal{C}}(s_i, j)\right\}_{j \in [\ell], j \neq i}$ are assigned the $\ell - 1$ closest permutations to s_i with higher probability information symbols mapped to closer permutations. For convenience, let us define the functions $\mathsf{skip}_i : \mathbb{N} \setminus \{i\} \to \mathbb{N}$

$$\mathsf{skip}_i(j) = \begin{cases} j, & j < i \\ j - 1, & j > i. \end{cases}$$

Thus, the average rewrite cost of a stored symbol $i \in [\ell]$, under any code, is lower-bounded by

$$\xi^{\mathcal{C}}(i) \geqslant \xi(i) \stackrel{\text{def}}{=} \sum_{j \in [\ell], j \neq i} p_j \delta_{\mathsf{skip}_i(j)}.$$

We continue by considering a specific intermediary *prefix-free code* that we denote by $\mathcal{Z}(i)$. Let it be induced by the prefix sequences $z_1^{(i)}, \ldots, z_\ell^{(i)}$. We require the following two properties:

P.1 For every $j \in [\ell]$, $j \neq i$, we require $\left|z_j^{(i)}\right| \leqslant 3\delta_{\mathsf{skip}_i(j)}$.

P.2 $\left|z_i^{(i)}\right| = 1$.

We also note that $\mathcal{Z}(i)$ is not necessarily a prefix-free code with minimal average codeword length.

Finally, let $\mathcal{A}$ be a prefix-free code that minimizes its average codeword length. Let $\mathcal{A}$ be induced by the prefix sequences $a_1, \ldots, a_\ell$, and let $s \in S_n$ be any state such that $\varphi^{\mathcal{A}}(s) = i$. Denote by $\xi^{\mathcal{A}}(s)$ the average rewrite cost of a rewrite operation under $\mathcal{A}$ starting from state s.

By the definition of $\mathcal{A}$ and $\mathcal{Z}(i)$ we have

$$\sum_{j \in [\ell]} p_j |a_j| \leqslant \sum_{j \in [\ell]} p_j \left|z_j^{(i)}\right|.$$

Since $|a_i| \geqslant 1 = \left|z_i^{(i)}\right|$ it follows that

$$\begin{aligned} \xi^{\mathcal{A}}(s) &= \sum_{j \in [\ell], j \neq i} p_j \alpha\left(s \to \mu^{\mathcal{A}}(s, j)\right) \\ &\leqslant \sum_{j \in [\ell], j \neq i} p_j |a_j| \leqslant \sum_{j \in [\ell], j \neq i} p_j \left|z_j^{(i)}\right| \\ &\leqslant \sum_{j \in [\ell], j \neq i} 3 p_j \delta_{\mathsf{skip}_i(j)} = 3\xi(i). \end{aligned}$$

Since the same argument works for every $s \in P_n(a_i)$, we can say that

$$\xi^{\mathcal{A}}(i) \leqslant 3\xi(i). \tag{2}$$

It is evident that the success of this proof strategy hinges on the existence of $\mathcal{Z}(i)$ for every $i \in [\ell]$, which we now turn to consider.

The following lemma is an application of the well-known Kraft–McMillan inequality [29].

Lemma 23: Let $r_1, r_2, \ldots, r_{n-1}$ be nonnegative integers. There exists a set of prefix sequences with exactly r_m prefix sequences of length m, for $1 \leqslant m \leqslant n - 1$ (i.e., there are r_m leaves in layer m of the code tree C), if and only if

$$\sum_{m=1}^{n-1} r_m \frac{(n-m)!}{n!} \leqslant 1.$$

Let us define the following sequence of integers:

$$r_m = \begin{cases} 1, & m = 1 \\ \left|\mathcal{S}^n_{m/3}\right|, & 2 \leqslant m \leqslant n-2, m \equiv 0 \pmod 3 \\ 0, & 2 \leqslant m \leqslant n-2, m \not\equiv 0 \pmod 3 \\ \frac{n!}{2} - \sum_{m'=1}^{n-2} r_{m'}, & m = n-1. \end{cases}$$

We first contend that they are all nonnegative. We only need to check r_{n-1} and indeed

$$\begin{aligned} r_{n-1} &= \frac{n!}{2} - \sum_{m=1}^{n-2} r_m = \frac{n!}{2} - \sum_{m=0}^{\left\lfloor \frac{n-2}{3} \right\rfloor} |\mathcal{S}^n_m| \\ &= \frac{n!}{2} - \left|\mathcal{B}^n_{\left\lfloor \frac{n-2}{3} \right\rfloor}\right| = \frac{n!}{2} - \frac{n!}{\left(n - \left\lfloor \frac{n-2}{3} \right\rfloor\right)!} \\ &\geqslant 0. \end{aligned}$$

It is also clear that

$$\sum_{m=1}^{n-1} r_m = \frac{n!}{2}.$$

In fact, in the following analysis, $r_1, r_2, \ldots, r_{n-1}$ represent a partition of the $\ell = n!/2$ alphabet letters.

Lemma 24: When $\ell = n!/2$, there exists a set of prefix sequences that contains exactly r_m prefix sequences of length m, for $m = 1, 2, \ldots, n - 1$.

Proof: Let us denote

$$R(n) = \sum_{m=1}^{n-1} r_m \frac{(n-m)!}{n!}. \tag{3}$$

When $n = 2,3,4,5,6,7$

$$R(n) = \frac{1}{2}, \frac{2}{3}, \frac{17}{24}, \frac{29}{40}, \frac{7}{10}, \frac{3377}{5040}$$

respectively. Thus, $R(n) \leqslant 1$ for all $2 \leqslant n \leqslant 7$. We now show that when $n \geqslant 8$, $R(n)$ monotonically decreases in n. Substituting $\{r_m\}$ into (3) we get

$$R(n) = \frac{1}{n} + \sum_{m=1}^{\lfloor \frac{n-2}{3} \rfloor} \frac{(n-3m)!}{(n-m+1)\cdot(n-m-1)!} + \frac{1}{2} - \frac{1}{\left(n - \lfloor \frac{n-2}{3} \rfloor\right)!}.$$

After some tedious rearrangement, for any integer $n \geqslant 8$

$$\begin{aligned} R(n) - R(n-1) &= -\frac{1}{n(n-1)} + \frac{(n \bmod 3)!(n - \lfloor n/3 \rfloor)}{(n - \lfloor n/3 \rfloor + 1)!} \\ &\quad - \sum_{m=1}^{\lfloor n/3 \rfloor - 1} \left(\frac{(n-1-3m)!}{(n-2-m)!} \cdot \right. \\ &\qquad \left. \cdot \frac{2nm - 2m^2 - 1}{((n-m)^2 - 1)(n-m)} \right) \\ &\leqslant -\frac{1}{n(n-1)} + \frac{1}{(n - \lfloor n/3 \rfloor - 1)!} < 0. \end{aligned}$$

Hence, $R(n)$ monotonically decreases for all $n \geqslant 8$ which immediately gives us $R(n) \leqslant 1$ for all $n \geqslant 2$. By Lemma 23, the proof is complete. □

We are now in a position to show the existence of $\mathcal{Z}(i)$, $i \in [\ell]$, for $\ell = n!/2$. By Lemma 24, let $z_1, z_2, \ldots, z_\ell$ be a list of prefix sequences, where exactly r_m of the sequences are of length m. Without loss of generality, assume

$$1 = |z_1| \leqslant |z_2| \leqslant \cdots \leqslant |z_\ell|.$$

Remember we also assume

$$p_1 \geqslant p_2 \geqslant \cdots \geqslant p_\ell.$$

We now define $\mathcal{Z}(i)$ to be the prefix-free code induced by the prefix sequences

$$\begin{array}{cccc} z_1^{(i)} = z_2 & z_2^{(i)} = z_3 & \cdots & z_{i-1}^{(i)} = z_i \\ & z_i^{(i)} = z_1 & & \\ z_{i+1}^{(i)} = z_{i+1} & \cdots & z_{\ell-1}^{(i)} = z_{\ell-1} & z_\ell^{(i)} = z_\ell \end{array}$$

that is, for all $j \in [\ell]$

$$z_j^{(i)} = \begin{cases} z_1, & j = i \\ z_{\mathsf{skip}_i(j)+1}, & j \in [\ell], j \neq i. \end{cases}$$

Note that for all $j \in [\ell]$, the prefix sequence $z_j^{(i)}$ represents the information symbol j, which is associated with the probability p_j in rewriting.

Lemma 25: The properties **P.1** and **P.2** hold for $\mathcal{Z}(i)$, $i \in [\ell]$.

Proof: Property **P.2** holds by definition, since $z_i^{(i)} = z_1$ whose length is set to $|z_1| = 1$. To prove property **P.1** holds, we first note that when $\ell = n!/2$, for all $r \in [n-2]$ there are exactly $|\mathcal{S}_r^n|$ indices j for which $\delta_j = r$. On the other hand, when $\ell = n!/2$, among the prefix sequences $z_2, \ldots, z_\ell$ we have $|\mathcal{S}_r^n|$ of them of length $3r$ when $3 \leqslant 3r \leqslant n-2$, and the rest of them are of length $n-1$. Intuitively speaking, we can map the $|\mathcal{S}_1^n|$ indices j for which $\delta_j = 1$ to distinct prefix sequences of length 3, the $|\mathcal{S}_2^n|$ indices j for which $\delta_j = 2$ to distinct prefix sequences of length 6, and so on.

Since the prefix sequences are arranged in ascending length order

$$|z_2| \leqslant |z_3| \leqslant \cdots \leqslant |z_\ell|$$

it follows that for every $j \in [\ell]$, $j \neq i$

$$\left| z_j^{(i)} \right| = \left| z_{\mathsf{skip}_i(j)+1} \right| \leqslant 3\delta_{\mathsf{skip}_i(j)}.$$

Hence, property **P.1** holds. □

We can now state the main theorem.

Theorem 26: Fix some $\ell \leqslant n!/2$ and let $\mathcal{A}$ be a prefix-free code over $[\ell]$ which minimizes its average codeword length. For any rewrite operation with initial stored information symbol $i \in [\ell]$

$$\xi^{\mathcal{A}}(i) \leqslant 3\xi(i)$$

i.e., the average cost of rewriting i under $\mathcal{A}$ is at most three times the lower bound.

Proof: Define $\ell' = n!/2$ and consider the input alphabet $[\ell']$ with input symbols being i.i.d. random variables where symbol $i \in [\ell']$ appears with probability p'_i. We set

$$p'_i = \begin{cases} p_i, & i \in [\ell] \\ 0, & i \in [\ell'] \setminus [\ell]. \end{cases}$$

Let $\mathcal{A}'$ be a prefix-free code over $[\ell']$ which minimizes its average codeword length.

A crucial observation is the following: $\xi(i)$, the lower bound on the average rewrite cost of symbol i, does depend on the probability distribution of the input symbols. Let us therefore distinguish between $\xi(i)$ over $[\ell]$, and $\xi'(i)$ over $[\ell']$. However, by definition, and by our choice of probability distribution over $[\ell']$

$$\xi(i) = \sum_{j \in [\ell], j \neq i} p_j \delta_{\mathsf{skip}_i(j)} = \sum_{j \in [\ell'], j \neq i} p'_j \delta_{\mathsf{skip}_i(j)} = \xi'(i)$$

for every $i \in [\ell]$. Since $\mathcal{A}'$ is a more restricted version of $\mathcal{A}$, it obviously follows that

$$\xi^{\mathcal{A}}(i) \leqslant \xi^{\mathcal{A}'}(i)$$

for every $i \in [\ell]$. By applying inequality (2), and since by Lemma 25, the code $\mathcal{Z}(i)$ exists over $[\ell']$, we get that

$$\xi^{\mathcal{A}}(i) \leqslant \xi^{\mathcal{A}'}(i) \leqslant 3\xi'(i) = 3\xi(i)$$

for all $i \in [\ell]$. □

Corollary 27: When $\ell \leqslant n!/2$, the average rewrite cost of a prefix-free code minimizing its average codeword length is at most three times that of an optimal code.

Proof: Since the approximation ratio of 3 holds for *every* rewrite operation (regardless of the initial state and its interpretation), it also holds for any average case. □

With a similar analysis, we can prove the following result:

Theorem 28: Fix some $\ell \leqslant n!/6$, $n \geqslant 4$, and let $\mathcal{A}$ be a prefix-free code over $[\ell]$ which minimizes its average codeword length. For any rewrite operation with initial stored information symbol $i \in [\ell]$

$$\xi^{\mathcal{A}}(i) \leqslant 2\xi(i)$$

i.e., the average cost of rewriting i under $\mathcal{A}$ is at most twice the lower bound.

Proof: See the Appendix. □

Corollary 29: When $\ell \leqslant n!/6$, $n \geqslant 4$, the average rewrite cost of a prefix-free code minimizing its average codeword length is at most twice that of an optimal code.

VI. Conclusion

In this paper, we present a new data storage scheme, *rank modulation*, for flash memories. We show several Gray code constructions for rank modulation, as well as data rewriting schemes. One important application of the Gray codes is the realization of logic multilevel cells. For data rewriting, an optimal code for the worst case performance is presented. It is also shown that to optimize the average rewrite cost, a prefix-free code can be constructed in polynomial time that approximates an optimal solution well under mild conditions. There are many open problems concerning rank modulation, such as the construction of error-correcting rank-modulation codes and codes for rewriting that are robust to uncertainties in the information symbol's probability distribution. Some of these problems have been addressed in some recent work [24].

Appendix

In this appendix, we prove Theorem 28. The general approach is similar to the proof of Theorem 26, so we only specify some details that are relatively important here.

We define the following sequence of numbers:

$$q_m = \begin{cases} 1, & m=1 \\ \left|\mathcal{S}^n_{m/2}\right|, & 2 \leqslant m \leqslant n-2,\ m \equiv 0 \pmod 2 \\ 0, & 2 \leqslant m \leqslant n-2,\ m \not\equiv 0 \pmod 2 \\ \frac{n!}{6} - \sum_{m'=1}^{n-2} q_{m'}, & m = n-1. \end{cases}$$

As before, we contend that they are all nonnegative. We only need to check q_{n-1} and indeed, for $n \geqslant 4$

$$\begin{aligned} q_{n-1} &= \frac{n!}{6} - \sum_{m=1}^{n-2} q_m = \frac{n!}{6} - \sum_{m=0}^{\lfloor \frac{n-2}{2} \rfloor} |\mathcal{S}^n_m| \\ &= \frac{n!}{6} - \left|\mathcal{B}^n_{\lfloor \frac{n-2}{2} \rfloor}\right| = \frac{n!}{6} - \frac{n!}{\left(n - \lfloor \frac{n-2}{2} \rfloor\right)!} \\ &\geqslant 0. \end{aligned}$$

We now prove the equivalent of Lemma 24.

Lemma 30: When $\ell = n!/6$, $n \geqslant 4$, there exists a set of prefix sequences that contains exactly q_m prefix sequences of length m, for $m = 1, 2, \ldots, n-1$.

Proof: Let us denote

$$Q(n) = \sum_{m=1}^{n-1} q_m \frac{(n-m)!}{n!}. \tag{4}$$

When $n = 4, 5, 6, 7, 8, 9, 10$

$$Q(n) = \frac{1}{2}, \frac{21}{40}, \frac{21}{40}, \frac{17}{35}, \frac{142}{315}, \frac{349}{840}, \frac{977}{2520},$$

respectively. Thus, $Q(n) \leqslant 1$ for all $4 \leqslant n \leqslant 10$. We now show that when $n \geqslant 11$, $Q(n)$ monotonically decreases in n. Substituting $\{q_m\}$ into (4) we get

$$Q(n) = \frac{1}{n} + \sum_{m=1}^{\lfloor \frac{n-2}{2} \rfloor} \frac{(n-2m)!}{(n-m+1)\cdot(n-m-1)!} + \frac{1}{6} - \frac{1}{\left(n - \lfloor \frac{n-2}{2} \rfloor\right)!}.$$

After some tedious rearrangement, for any integer $n \geqslant 11$

$$\begin{aligned} &Q(n) - Q(n-1) \\ &\quad = -\frac{1}{n(n-1)} + \frac{(2 - (n \bmod 2))!(n - \lfloor n/2 \rfloor)}{(n - \lfloor n/2 \rfloor + 1)!} \\ &\qquad - \sum_{m=1}^{\lfloor (n-1)/2 \rfloor - 1} \left(\frac{(n-1-2m)!}{(n-2-m)!} \cdot \right. \\ &\qquad\qquad \left. \cdot \frac{nm - m^2 - 1}{((n-m)^2 - 1)(n-m)} \right) \\ &\quad \leqslant -\frac{1}{n(n-1)} + \frac{1}{(n - \lfloor n/2 \rfloor - 1)!} < 0. \end{aligned}$$

Hence, $Q(n)$ monotonically decreases for all $n \geqslant 11$ which immediately gives us $Q(n) \leqslant 1$ for all $n \geqslant 4$. By Lemma 23, the proof is complete. □

The remaining lemmas comprising the rest of the proof procedure are similar to those of Section V-B.

Acknowledgment

The authors would like to thank the anonymous reviewers, whose comments helped improve the presentation of the paper.

References

[1] M. B. Baer, "Optimal prefix codes for infinite alphabets with nonlinear costs," *IEEE Trans. Inf. Theory*, vol. 54, no. 3, pp. 1273–1286, Mar. 2008.

[2] A. Bandyopadhyay, G. Serrano, and P. Hasler, "Programming analog computational memory elements to 0.2% accuracy over 3.5 decades using a predictive method," in *Proc. IEEE Int. Symp. Circuits and Systems*, Kobe, Japan, May 2005, pp. 2148–2151.

[3] T. Berger, F. Jelinek, and J. K. Wolf, "Permutation codes for sources," *IEEE Trans. Inf. Theory*, vol. IT-18, no. 1, pp. 160–169, Jan. 1972.

[4] V. Bohossian, A. Jiang, and J. Bruck, "Buffer coding for asymmetric multi-level memory," in *Proc. IEEE Int. Symp. Information Theory (ISIT2007)*, Nice, France, Jun. 2007, pp. 1186–1190.

[5] P. Cappelletti and A. Modelli, "Flash memory reliability," in *Flash Memories*, P. Cappelletti, C. Golla, P. Olivo, and E. Zanoni, Eds. Amsterdam, The Netherlands: Kluwer, 1999, pp. 399–441.

[6] 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.

[7] P. Dietz, *Optimal Algorithms for List Indexing and Subset Rank*. London, U.K.: Springer-Verlag, 1989.

[8] B. Eitan and A. Roy, "Binary and multilevel flash cells," in *Flash Memories*, P. Cappelletti, C. Golla, P. Olivo, and E. Zanoni, Eds. Amsterdam, The Netherlands: Kluwer, 1999, pp. 91–152.

[9] T. Etzion, Oct. 2007, personal communication.

[10] A. Fiat and A. Shamir, "Generalized write-once memories," *IEEE Trans. Inf. Theory*, vol. IT-30, no. 3, pp. 470–480, May 1984.
[11] 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.
[12] M. J. Golin and G. Rote, "A dynamic programming algorithm for constructing optimal prefix-free codes with unequal letter costs," *IEEE Trans. Inf. Theory*, vol. 44, no. 5, pp. 1770–1781, Sep. 1998.
[13] F. Gray, "Pulse Code Communication," U.S. Patent 2632058, Mar. 1953.
[14] M. Grossi, M. Lanzoni, and B. Ricco, "Program schemes for multilevel flash memories," *Proc. IEEE*, vol. 91, no. 4, pp. 594–601, Apr. 2003.
[15] M. Hall, Jr and D. E. Knuth, "Combinatorial analysis and computers," *Amer. Math. Monthly*, vol. 72, no. 2, pp. 21–28, 1965.
[16] 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.
[17] C. D. Heegard, "On the capacity of permanent memory," *IEEE Trans. Inf. Theory*, vol. IT-31, no. 1, pp. 34–42, Jan. 1985.
[18] C. D. Heegard and A. A. El Gamal, "On the capacity of computer memory with defects," *IEEE Trans. Inf. Theory*, vol. IT-29, no. 5, pp. 731–739, Sep. 1983.
[19] D. A. Huffman, "A method for the construction of minimum-redundancy codes," *Proc. IRA*, vol. 40, no. 9, pp. 1098–1101, Sep. 1952.
[20] F. K. Hwang, "Generalized Huffman trees," *SIAM J. Appl. Math.*, vol. 37, no. 1, pp. 124–127, 1979.
[21] A. Jiang, "On the generalization of error-correcting WOM codes," in *Proc. IEEE Int. Symp. Information Theory (ISIT2007)*, Nice, France, Jun. 2007, pp. 1391–1395.
[22] A. Jiang, V. Bohossian, and J. Bruck, "Floating codes for joint information storage in write asymmetric memories," in *Proc. IEEE Int. Symp. Information Theory (ISIT2007)*, Nice, France, Jun. 2007, pp. 1166–1170.
[23] A. Jiang and J. Bruck, "Joint coding for flash memory storage," in *Proc. IEEE Int. Symp. Information Theory (ISIT2008)*, Toronto, ON, Canada, Jul. 2008, pp. 1741–1745.
[24] A. Jiang, M. Schwartz, and J. Bruck, "Error-correcting codes for rank modulation," in *Proc. IEEE Int. Symp. Information Theory (ISIT2008)*, Toronto, ON, Canada, Jul. 2008, pp. 1736–1740.
[25] D. E. Knuth, *The Art of Computer Programming Volume 3: Sorting and Searching*, 2nd ed. Reading, MA: Addison-Wesley, 1998.
[26] C. A. Laisant, "Sur la numération factorielle, application aux permutations," *Bull. Société Mathématique de France*, vol. 16, pp. 176–183.
[27] D. H. Lehmer, "Teaching combinatorial tricks to a computer," in *Proc. Symp. Applied Mathematics and Combinatorial Analysis*, 1960, vol. 10, pp. 179–193.
[28] M. Mares and M. Straka, "Linear-time ranking of permutations," *Algorithms-ESA*, pp. 187–193, 2007.
[29] B. McMillan, "Two inequalities implied by unique decipherability," *IEEE Trans. Inf. Theory*, vol. 2, no. 4, pp. 115–116, 1956.
[30] W. Myrvold and F. Ruskey, "Ranking and unranking permutations in linear time," *Inf. Process. Lett.*, vol. 79, no. 6, pp. 281–284, 2001.
[31] H. Nobukata, S. Takagi, K. Hiraga, T. Ohgishi, M. Miyashita, K. Kamimura, S. Hiramatsu, K. Sakai, T. Ishida, H. Arakawa, M. Itoh, I. Naiki, and M. Noda, "A 144-Mb, eight-level nand flash memory with optimized pulsewidth programming," *IEEE J. Solid-State Circuits*, vol. 35, no. 5, pp. 682–690, May 2000.
[32] R. L. Rivest and A. Shamir, "How to reuse a write-once memory," *Inf. Contr.*, vol. 55, pp. 1–19, 1982.
[33] C. D. Savage, "A survey of combinatorial gray codes," *SIAM Rev.*, vol. 39, no. 4, pp. 605–629, Dec. 1997.
[34] R. Sedgewick, "Permutation generation methods," *Comput. Surv.*, vol. 9, no. 2, pp. 137–164, Jun. 1977.
[35] D. Slepian, "Permutation modulation," *Proc. IEEE*, vol. 53, no. 3, pp. 228–236, Mar. 1965.
[36] J. K. Wolf, A. D. Wyner, J. Ziv, and J. Körner, "Coding for a write-once memory," *AT&T Bell Labs. Tech. J.*, vol. 63, no. 6, pp. 1089–1112, 1984.
[37] E. Yaakobi, P. H. Siegel, and J. K. Wolf, "Buffer codes for multi-level flash memory," presented at the Poster Session of the IEEE Int. Symp. Information Theory, Toronto, ON, Canada, Jul. 2008.

Anxiao (Andrew) Jiang (S'00)-M'04) 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, in 2000 and 2004, respectively.

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

Robert Mateescu (M'08) received the B.S. degree in computer science and mathematics from the University of Bucharest, Bucharest, Romania, in 1997 and the M.S. and Ph.D. degrees in information and computer science from the University of California, Irvine, in 2003 and 2007, respectively.

He is currently a Postdoctoral Scholar in Electrical Engineering at the California Institute of Technology, Pasadena. His research interests include algorithms for the representation and inference of information.

Moshe Schwartz (M'03) was born in Israel in 1975. He received the B.A., 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 Postdoctoral Researcher in the Department of Electrical and Computer Engineering, University of California, San Diego, La Jolla, and a Postdoctoral Researcher in the Department of Electrical Engineering, California Institute of Technology, Pasadena. He now holds a position with the Department of Electrical and Computer Engineering, Ben-Gurion University, Beer-Sheva, Israel. 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 (Caltech), Pasadena. His research focuses on information theory and systems and the theory biological networks. He has an extensive industrial experience. He worked at IBM Research where he participated in the design and implementation of the first IBM parallel computer. He was a cofounder and Chairman of Rainfinity, a spinoff company from Caltech that focused on software products for management of network information storage systems.

Dr. Bruck received the National Science Foundation Young Investigator award, the Sloan fellowship, and the 2005 S. A. Schelkunoff Transactions prize paper award from the IEEE Antennas and Propagation society.