Rank Modulation for Flash Memories

Anxiao (Andrew) Jiang[*] **Robert Mateescu**[†] **Moshe Schwartz**[‡] **Jehoshua Bruck**[†]

[*]Department of Computer Science
Texas A&M University
College Station, TX 77843, U.S.A.
ajiang@cs.tamu.edu

[†]California Institute of Technology
1200 E California Blvd., Mail Code 136-93
Pasadena, CA 91125, U.S.A.
{*mateescu,bruck*}*@paradise.caltech.edu*

[‡]Electrical and Computer Engineering
Ben-Gurion University
Beer Sheva 84105, Israel
schwartz@ee.bgu.ac.il

***Abstract*—We explore a novel data representation scheme for multi-level 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. We also investigate optimal rewriting schemes for translating arbitrary input alphabet into n-cell states which minimize the number of programming operations.

I. Introduction

Flash memory is a non-volatile memory technology that is both electrically programmable and electrically erasable. Its reliability, high storage density, and relatively low cost have made flash memory a dominant non-volatile 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 except for the single cell to be erased.

To keep up with the ever-growing demand for denser storage, the *multi-level flash cell* concept is used to increase the number of stored bits in a cell [4]. Instead of the usual single-bit flash memories, where each cell is in one of two states (erased/programmed), each multi-level 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 multi-level flash memories are a topic of significant research and design efforts [12], [7], [1]. 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 multi-level 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* [3], 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 multi-level 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 modulation, and data rewriting schemes. Error-correcting codes for rank modulation are the subject of a companion paper [11]. The Gray code [6] 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

This work was supported in part by the Caltech Lee Center for Advanced Networking.

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 which are minimal with respect to some cost function, thus creating a traversal of the state space which is minimal in total cost. For a comprehensive survey of combinatorial Gray codes, the reader is referred to [13].

Some of the Gray code constructions we describe 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 [13], and the more specific [14] (and references therein). In [14] the transitions we use in this paper are called "nested cycling" and the algorithms cited there produce lists which are not Gray codes since some of the permutations repeat, which makes the algorithms inefficient.

We also investigate rewriting schemes for rank modulation. Since erasing/reprogramming cells is expensive, it is very important to maximize the number of times data can be rewritten between two erasure operations [2], [9], [10]. 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 average-case performance.

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 presents a rewriting scheme that is optimal for the worst-case performance, and a dynamic programming algorithm to find the optimal prefix code for the average cost of rewriting; Section V 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 i-th 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 codes *length n Rank Modulation Gray Codes* (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 first column.

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 $t^{(1)}, t^{(2)}, \ldots, t^{((n-1)!)}$ and where $t^{((n-1)!)} = t_2$, i.e., a "push-to-the-top" operation on the second element in the 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 $t^{(1)}, t^{(2)}, \ldots, t^{((n-1)!-1)}$ to get a list of $(n-1)!$ permutations 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 $t^{((n-1)!)} = t_2$, 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 $t^{(1)}, t^{(2)}, \ldots, t^{((n-1)!-1)}$ again to create the entire second block. We repeat this process $n-1$ times, i.e., use the sequence of transitions $t^{(1)}, t^{(2)}, \ldots, t^{((n-1)!-1)}, t_n$ a total of $n-1$ times to construct $n-1$ blocks, each containing $(n-1)!$ permutations.

The following two simple lemmas are given without proof.

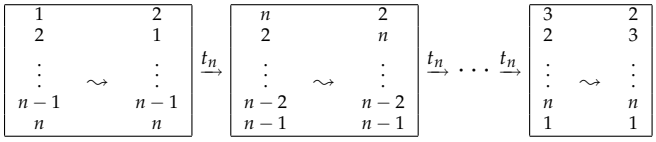
Lemma 2. *In any block, the last element of all the permutations is constant. The list of last elements in the blocks constructed is $n, n-1, \ldots, 3, 1$. The element 2 is never a last element.*

Lemma 3. *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.*

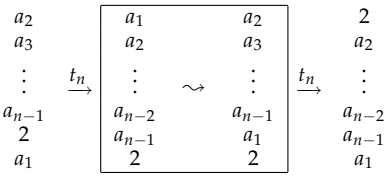
Combining the two lemmas above, the $n-1$ blocks constructed so far form a cyclic but not complete n-RMGC, that we call C', which may be schematically described as follows

[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.

(where each box represents a single block, and $\rightsquigarrow$ denotes the sequence of transitions $t^{(1)},\ldots,t^{((n-1)!-1)}$):



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 $t^{(1)},\ldots,t^{((n-1)!)}$ such that its last transition is t_{n-1}[2]. For convenience we denote the rotated sequence by $\tau^{(1)},\ldots,\tau^{(n-1)!}$, where $\tau^{(n-1)!}=t_{n-1}$. The first permutation in the block is $[a_1,a_2,\ldots,a_{n-1},2]$, and the last one is $[a_2,\ldots,a_{n-1},a_1,2]$. In C' we find a transition of the following form: $[a_2,\ldots,a_{n-1},2,a_1]\xrightarrow{t_{n-1}}[2,a_2,\ldots,a_{n-1},a_1]$. Such a transition must surely exist since C' is cyclic, it contains permutations in which 2 is next to last and some in which it is not, it does not contain permutations in which 2 is last, and so it follows that at some point in C', the element 2 is next to last and is then pushed by t_{n-1} to the front. At this transition we split C' and insert C'' as follows:



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 $t^{(1)},\ldots,t^{((n-1)!-1)}$ contains at least one occurrence of t_2, and is replicated $n-1$ times, $n\geqslant 3$. We therefore reach the following conclusion:

Corollary 4. *For every $n\geqslant 2$ there exists a cyclic and complete n-RMGC.*

The 3-RMGC shown in Example 1 is the result of this construction for $n=3$.

III. Balanced n-RMGCs

A. Definition and Construction

It is sometimes the case that due to precision constraints in the charge placement mechanism, the actual possible charge levels in flash memory cells are discrete. Thus, we define the function $c_i:\mathbb{N}\to\mathbb{N}$, where $c_i(p)$ is the charge level of the i-th cell after the p-th programming cycle. It follows that if we use transition t_j in the p-th programming cycle and the i-th cell is, at the time, j-th 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, $c_i(p)=\max_k\{c_k(p-1)\}+1$.

[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.

The *jump* in the p-th round is defined as $c_i(p)-c_i(p-1)$, assuming the i-th 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 maximal jump during the transitions dictated by the code. It is easy to see that the lowest possible jump cost in an optimal RMGC is at least $n+1$, for $n\geqslant 3$. That is because 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 jump cost $n+1$ a *balanced n-RMGC*. We now show a construction that turns any $(n-1)$-RMGC into a balanced n-RMGC while retaining properties such as being cyclic or complete.

Theorem 5. *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 $\mathcal{C}_n$, that is cyclic, complete and* balanced*:*

$$\text{For } k\in\{1,\ldots,n!\},\ t_k=\begin{cases}t_{n-i_{\lceil k/n\rceil}+1} & \text{, if } k\equiv 1 \pmod n\\ t_n & \text{, otherwise.}\end{cases}$$

Proof: Let us define the abstract transition $\overrightarrow{t_i}$, $2\leqslant i\leqslant n$, that pushes *to the bottom* the i-th 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. If the initial permutation is $[1,2,\ldots,n]$, the element 1 is fixed as the first element of the resulting permutations.

The $(n-1)!$ permutations of $[n]$ produced by $\overrightarrow{t_{i_1}},\ldots,\overrightarrow{t_{i_{(n-1)!}}}$, also have the property of being representatives of the $(n-1)!$ distinct orbits of the permutations of $[n]$ under the operation t_n. That means that there are no two permutations which are cyclic shifts of each other, 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 $\overrightarrow{t_i}$. Every transition of the form t_{n-i+1}, $i\neq n$, moves us to a different orbit, 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 $\mathcal{C}_n$ based on C_{n-1} is:

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

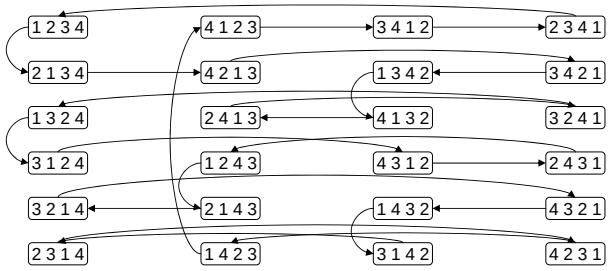


Figure 1. Balanced 4-RMGC

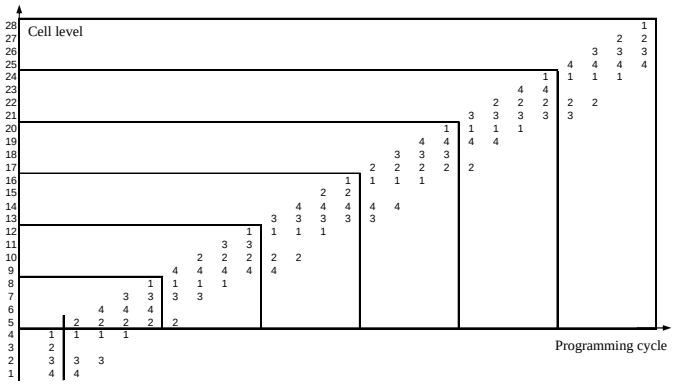


Figure 2. Balanced 4-RMGC growth

The code $\mathcal{C}_n$ 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. ■

We can use Theorem 5 to recursively construct all the supporting j-RMGCs, $j \in \{n-1, \ldots, 2\}$, with the basis of the recursion being the the 2-RMGC: $[1,2], [2,1]$.

Corollary 6. *For any $n \geqslant 2$, there exists a cyclic, complete and balanced n-RMGC.*

A similar construction, but using a more involved second order recursion, was later suggested by Etzion [5].

Example 7. *Figure* 1 *shows the transitions of a recursive, balanced* 4*-RMGC. The permutations are represented in an n by $(n-1)!$ matrix. Each row is an orbit generated by t_n. Each column has the last element fixed. The transitions between rows occur when* 1 *is the top (leftmost) element. These transitions are defined recursively, by a balanced* 3*-RMGC over the set $\{2,3,4\}$ (where the top element is now the rightmost one): [2,3,4],[3,4,2],[3,2,4],[2,4,3],[4,3,2],[4,2,3]. They are $\overrightarrow{t_3}, \overrightarrow{t_2}, \overrightarrow{t_3}, \overrightarrow{t_3}, \overrightarrow{t_2}, \overrightarrow{t_3}$. This is the cycle from Example* 1*, with relabeled cells, and starting with the third column.*

Example 8. *Figure* 2 *shows the cell levels of the recursive balanced* 4*-RMGC for each programming cycle.*

There is another asymptotic measure by which the construction for the recursive balanced n-RMGCs is optimal. 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 of the form "what is the i-th highest charged cell?".

Suppose we were to use the recursive construction of Theorem 5 to generate a cyclic, complete and balanced n-RMGC with a starting permutation $[1,2,\ldots,n]$. A fraction $\frac{n-1}{n}$ of the transitions are t_n, and these occur whenever the cell 1 is not the highest charged one. Of the cases where 1 is highest charged, by recursion, a fraction $\frac{n-2}{n-1}$ of the transitions are determined by just one query, and so on. At the basis of the recursion, permutations over two elements require zero queries. Thus, the total number of queries is $\sum_{i=3}^{n} i!$. Since $\lim_{n\to\infty} \frac{\sum_{i=3}^{n} i!}{n!} = 1$, the asymptotic average number of steps to generate the next permutation is just 1.

B. Rank and Unrank Functions

We define here the two inverse functions that associate each permutation with a number from 1 to $n!$. The *rank* and *unrank* functions for the balanced n-RMGC follow the recursive construction. Using the representation of Example 7, the *rank* function can be computed by determining the row and the cyclic offset from the first element encountered in the current row. The row is determined by the recursive application of *rank* to the $(n-1)$-RMGC, while the cyclic offset is determined by the position of 1 in the current permutation.

We change the first permutation from $[1,2,\ldots,n]$ to $[n,1,3,\ldots,4,2]$, where n is first, and the odd numbers are written in increasing order from left to right, while the even numbers are written in increasing order from right to left. The *rank* function is given by the following recursive procedure:

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

input : $n \in \mathbb{N}, n \geqslant 2$; permutation $[a_1,\ldots,a_n]$
output : Index of $([a_1,\ldots,a_n])$ in the recursive balanced n-RMGC starting with $[n,1,3,\ldots,4,2]$

```
if n = 2 then
    if a_1 > a_2 then
        return 1
    else
        return 2
else
    Find i such that a_i = min{a_1, ..., a_n}
    return
    i - 1 + n [Rank(n - 1, [a_n, ..., a_{i+1}, a_{i-1}, ..., a_1]) - 1]
```

The *rank* function can also be computed in a non-recursive way, as follows. Let $count(i), 1 \leqslant i \leqslant n$ be the number of elements greater than i that lie to the left of i, if i is odd (and to the right of i if i is even). Let $pos_i = count(i)$ if $count(i) \neq 0$, and $pos_i = n-i+1$ if $count(i) = 0$ (this defines the position in row).

If $\pi = [a_1,\ldots,a_n]$, let $\pi^{i:n}$ be the permutation defined by the subset $\{i,\ldots,n\}$, taken in cyclic order, beginning with i and reading to the left if i is odd, and to the right if i is even; For example $[3,1,4,2]^{2:4} = [3,2,4]$, namely $[3,2,4]$ is the permutation that generates the cyclic orbit represented by the row of $[3,1,4,2]$ in the recursive balanced 4-RMGC.

We have $rank(\pi) = pos_1 + n \cdot (rank(\pi^{2:n}) - 1)$. Unfolding the recursive expression, we get: $rank(\pi) = 1 + (pos_1 - 1) + n(pos_2 - 1) + n(n-1)(pos_3 - 1) + \ldots + n(n-1)\cdots 3(pos_{n-1} - 1) = 1 + \sum_{i=1}^{n-1} \left[\prod_{j=0}^{i-2}(n-j)\right](pos_i - 1)$.

The *unrank* function maps a number k, $1 \leqslant k \leqslant n!$, to a permutation from the recursive balanced n-RMGC that starts with $[n, 1, 3, \ldots, 4, 2]$. The function *unrank* can be computed by a recursive procedure that first computes the position of 1 (namely, the cyclic offset in the row), and then recurses to determine the position of the other elements. The position of 1 is 1 (the first element from left, namely the permutation is the last one traversed in its row) if $k \equiv 0 \mod n$; otherwise it is $(k \mod n) + 1$. The position of 2 is determined by making $k \leftarrow \lceil k/n \rceil$, and counting its position from right to left, while ignoring the position already occupied by 1. The position of the other elements is determined recursively in the same manner. The *unrank* function is given by the following procedure:

Function `Unrank`(k, n)

```
input  : k, n ∈ ℕ, 1 ⩽ k ⩽ n!
output : The k-th permutation of the recursive balanced
         n-RMGC starting with [n, 1, 3, ..., 4, 2]
Initialize [a_1, ..., a_n] ← [0, ..., 0]
for i ← 1 to n − 1 do
    p ← Position(k, n − i + 1)
    if i is odd then
        m ← 0
        for j ← 1 to p do
            m ← m + 1
            while a_m ≠ 0 do
                m ← m + 1
    else
        m ← n + 1
        for j ← 1 to p do
            m ← m − 1
            while a_m ≠ 0 do
                m ← m − 1
    a_m ← i
    k ← ⌈k/n⌉
Find the remaining i for which a_i = 0 and set a_i ← n
return [a_1, ..., a_n]
```

Function `Position`(k, n)

```
input  : k, n ∈ ℕ, 1 ⩽ k ⩽ n!
output : Position of element 1 in the k-th permutation of the
         recursive balanced n-RMGC starting with
         [n, 1, 3, ..., 4, 2]
if k ≡ 0 mod n then
    return 1
else
    return (k mod n) + 1
```

IV. Rewriting with Rank Modulation Codes

In this section, we study coding schemes for rewriting data in flash memories. When the data stored using a rank modulation code needs to be modified, the flash memory can increase some cells' charge levels so that the updated cell state represents the new data. The highest charge level increases with each rewriting operation. When it reaches the maximum possible charge level, the next rewriting leads to the block erasure and reprogramming. Since block erasure/reprogramming is expensive – it not only is time/power consuming, but also reduces the reliability and longevity of flash memories, – it is very important to maximize the number of times that data can be rewritten between two block erasure operations [2], [9], [10]. For rank modulation, the key is to minimize the number of cells whose charge levels need to be pushed to the top during the rewriting operation. We investigate schemes that achieve this objective.

In order to discuss rewriting, we first need to define a decoding scheme. It is often the case that the alphabet size used by the user to input data and read stored information differs from the alphabet size used as internal representation. In our case, data is stored internally in one of $n!$ different permutations. Let us assume the user alphabet is $Q = \{1, 2, \ldots, q\}$. A *decoding scheme* is a function $D : S \to Q$ mapping internal states to symbols from the user alphabet.

Suppose the current internal state is $s_1 \in S$ and the user inputs a new symbol $\alpha \in Q$. A *rewriting operation given* α is now defined as moving from state $s_1 \in S$ to state $s_2 \in S$ such that $D(s_2) = \alpha$. It should be noted that if $D(s_1) = \alpha$ then s_2 may be equal to s_1, i.e., the rewriting operation is degenerate and does nothing. The *cost* of the rewriting operation is the minimal number of atomic transitions from T (i.e., the number of "push-to-the-top" operations) required to move from state s_1 to state s_2.

It is now obvious that rewriting operations requiring a large number of transitions are undesirable both because they promote charge-level saturation, and because they take longer to perform. In the following section we first present a decoding scheme that strictly optimizes the rewriting performance for the worst case. Then, we extend the construction to optimize the average rewrite performance with constant approximation ratios.

A. Optimal Decoding Scheme for Rewriting

We start by presenting a lower bound on the cost of a single rewriting operation. First, we define a few terms. Define the *transition graph* $G = (V, E)$ as a directed graph with $V = S_n$, i.e., with $n!$ vertices representing the permutations in S_n. There is a directed edge $u \to v$ if $v = t(u)$ for some $t \in T$, i.e., we can obtain v from u by a single "push-to-the-top" operation. We can see that G is a regular digraph: every vertex has $n-1$ incoming edges and $n-1$ outgoing edges.

For two vertices $u, v \in V$, we define the directed distance $d(u, v)$ as the number of edges in the shortest directed path from u to v. Clearly, $0 \leqslant d(u, v) \leqslant n-1$ for any $u, v \in V$. Given a vertex $u \in V$ and an integer r (here $0 \leqslant r \leqslant n-1$), we define the *ball* $\mathcal{B}_r(u)$ as $\mathcal{B}_r(u) = \{v \in V \mid d(u, v) \leqslant r\}$, and define the *sphere* $\mathcal{S}_r(u)$ as $\mathcal{S}_r(u) = \{v \in V \mid d(u, v) = r\}$. Clearly, $\mathcal{B}_r(u) = \bigcup_{0 \leqslant i \leqslant r} \mathcal{S}_r(u)$.

Lemma 9. *For any $u \in V$ and $0 \leqslant r \leqslant n-1$,*

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

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

Proof: A vertex v is in $\mathcal{B}_r(u)$ if and only if we can obtain it at most r transitions. There are $n(n-1)\cdots(n-r+1)$ ways to do so, hence $|\mathcal{B}_r(u)| = \frac{n!}{(n-r)!}$. Since $\mathcal{B}_r(u) = \bigcup_{0 \leqslant i \leqslant r} S_u^r$ and the spheres are disjoint, the rest of the conclusion follows. ∎

Let ρ denote the smallest integer such that $|\mathcal{B}_\rho(u)| \geqslant q$. Note that ρ is independent of u. The following lemma presents a bound on the rewriting cost.

Lemma 10. *For any decoding scheme and any current state, there exists $\alpha \in Q$ such that the cost of a rewriting operation given α is at least ρ.*

Proof: Fix a decoding scheme D, and let s be the current state. By our definition of ρ we have $|\mathcal{B}_{\rho-1}(s)| < q$. It follows that we can choose $\alpha \in Q \setminus \{D(s') \mid s' \in \mathcal{B}_{\rho-1}(s)\}$. Clearly, a rewriting operation given α requires at least ρ transitions since there is no $s' \in \mathcal{B}_{\rho-1}(s)$ such that $D(s') = \alpha$. ∎

Next, we present a code construction. It will be shown to be optimal.

Construction 11. *Divide the $n!$ states S_n into $\frac{n!}{(n-\rho)!}$ sets, where two states are in the same set if and only if their ρ top-charged cells are the same. Among the sets, choose q sets and map them to the q symbols of Q arbitrarily. The other $\frac{n!}{(n-\rho)!} - q$ sets need not represent any symbol.*

Example 12. *Let $n = 3$ and $q = 3$. Since we have $|\mathcal{B}_1(u)| = 3$, it follows that $\rho = 1$. We divide the $n! = 6$ states into $\frac{n!}{(n-\rho)!} = 3$ sets which induce the decoding function: $\{[1,2,3],[1,3,2]\} \mapsto 1$, $\{[2,1,3],[2,3,1]\} \mapsto 2$, and $\{[3,1,2],[3,2,1]\} \mapsto 3$. The two states in the set are decoded to the same symbol from Q. The cost of any rewrite operation is at most 1.*

Since the top ρ cells of a state uniquely determine the decoded symbol, any rewriting operation costs at most ρ transitions to replace the top ρ cells. By Lemma 10, we obtain the optimality of the scheme:

Theorem 13. *The decoding scheme presented in Construction 11 guarantees a rewriting cost of at most ρ transitions, and is therefore optimal.*

B. Optimizing the Average Cost of Rewriting

The scheme presented in the previous section optimizes the worst-case performance. In practice, however, if the probabilities with which the input symbol takes values from its alphabet are known, it is also important to study schemes that optimize the average cost of rewriting.

Let us assume that for each rewrite (including writing the initial value), the input symbol is drawn i.i.d. from $Q = \{1, 2, \ldots, q\}$ with probability p_i to get symbol $i \in Q$. We study decoding schemes that optimize the average cost of rewriting. Depending on the probabilities $\{p_i\}$, the optimal code may be quite complex. The code design problem is closely related to the facility-location problem, which is NP hard.

In this paper, we present a prefix code that is optimal in terms of its own design objective. The design objective is also a bound for the optimal performance of all rankmodulation codes. Furthermore, we will prove that at least when $q \leqslant n!/2$, the prefix code is a 3-approximation of any optimal solution, that is, the cost of the prefix code is at most three times the cost of any optimal rank modulation code. We will also show that when $q \leqslant n!/6$, the prefix code is a 2-approximation of an optimal solution.

The prefix code we propose consists of q codewords of "variable lengths", which represent the q values of the stored input. Each codeword is a prefix of a permutation from S_n. No codeword is allowed to be the prefix of another codeword. Let $a = [a_1, a_2, \ldots, a_i]$ be a generic codeword that represents the value $\alpha \in Q$. For a state $s \in S_n$, if a is a prefix of s then we set $D(s) = \alpha$. Due to the prefix-free property, the decoding function is well-defined.

A prefix 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. The 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. 3(a). A prefix code corresponds to a subtree C of T (see Fig. 3(b) for an example). Every codeword is mapped to a leaf, and the codeword is the same as the labels on the path from the root to the leaf.

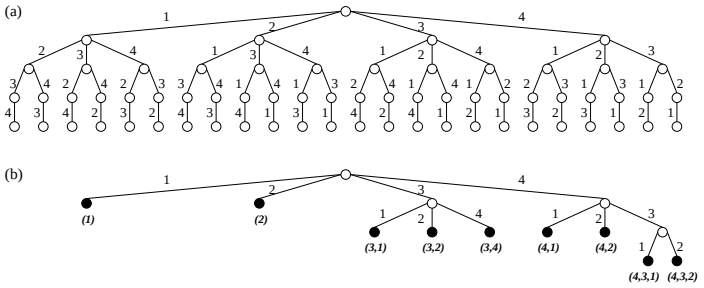


Figure 3. Prefix rank modulation code for $n = 4$ and $q = 9$. (a) The full permutation tree T. (b) A prefix code represented by a subtree C of T. The leaves represent the codewords, which are the labels beside the leaves.

For $i \in Q$, let c_i denote the codeword representing i, and let $|c_i|$ denote its length. For example, the codewords in Fig. 3(b) have minimum length of 1 and maximum length of 3. The average codeword length is defined as $\sum_{i=1}^{q} p_i |c_i|$. Our objective is to design a code that *minimizes the average codeword length.*

The optimal prefix code cannot be constructed with a greedy algorithm like the Huffman code and its extensions [8],

because the vertex degrees in the code tree C are unknown initially. We present a dynamic programming algorithm of time complexity $O(nq^4)$ to construct the optimal code. Note that without loss of generality, we can assume that a codeword's length is at most $n-1$.

The algorithm computes a set of functions $\mathsf{opt}_i(\ell,m)$, for $i = 1,2,\ldots,n-1$, $\ell = 0,1,\ldots,q$, and $m = 0,1,\ldots,\min\{q,n!/(n-i)!\}$. We interpret the meaning of $\mathsf{opt}_i(\ell,m)$ as follows. We take a subtree of T that contains the root. The subtree has exactly ℓ leaves in the layers $i,i+1,\ldots,n-1$. It also has at most m vertices in the layer i. We let the ℓ leaves represent the ℓ input values from Q with the lowest probabilities p_j: the further the leaf is from the root, the lower the corresponding probability is. Those leaves are also ℓ codewords, 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 $\mathsf{opt}_i(\ell,m)$. In other words, $\mathsf{opt}_i(\ell,m)$ is the minimum average codeword length when we assign a subset of codewords to a subtree of T (in a specific way). Clearly, the *minimum average codeword length* of a prefix code equals $\mathsf{opt}_1(q,n)$.

Without loss of generality, let us assume that $p_1 \leqslant p_2 \leqslant \cdots \leqslant p_q$. It is easily seen that the following recursion holds:

$$\mathsf{opt}_i(\ell,m) = \begin{cases} (n-1)\sum_{k=1}^{\ell} p_k & i = n-1,\ m \geqslant \ell > 0 \\ 0 & i > 1,\ \ell = 0 \\ \min_{0\leqslant j\leqslant \min\{\ell,m\}} & \\ \{\mathsf{opt}_{i+1}(\ell-j, & \\ \min\{q,(m-j)\cdot & \\ (n-i)\})+ & \\ \sum_{k=\ell-j+1}^{\ell} i p_k\} & i < n-1,\ \ell,m > 0 \end{cases}$$

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

The algorithm first computes $\mathsf{opt}_{n-1}(\ell,m)$, then $\mathsf{opt}_{n-2}(\ell,m)$, and so on until finally computing $\mathsf{opt}_1(q,n)$, by using the above recursions. Given this value, it is straightforward to determine in the optimal code, how many codewords 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 $O(nq^4)$.

We can use the prefix code for rewriting in the following way: to change the stored value to $i \in Q$, we raise at most $|c_i|$ cells so that the $|c_i|$ top-ranked cells are the same as the codeword c_i. Since the probability that the variable is i after *every* rewrite equals p_i, the average codeword length of the optimal prefix code is an upper bound for the average rewriting cost of all optimal rank modulation codes.

We obviously have $q \leqslant n!$. When $q = n!$, the code design becomes trivial. In practice, the scenario where $q \leqslant n!/2$ is important, and the optimal code design can be complex. We prove in the following that when $q \leqslant n!/2$, the average rewriting cost of the optimal prefix code is at most three times that of any rank modulation code, thus making it a 3-approximation solution.

The general idea of the proof is as follows. Let $i \in Q$ denote the value of the stored data at a given moment. Let $s_i \in S_n$ denote the cell state at that moment, and let $s_1, s_2, \cdots, s_{i-1}, s_{i+1}, \cdots, s_{q-1}, s_q$ denote the $q-1$ cell states whose distance from s_i in the *transition graph*, $d(s_i,s_j)$ for $1 \leqslant j \leqslant q$ and $j \neq i$, are the smallest ones. $d(s_i,s_j)$ is the rewriting cost (i.e., the number of cell charge levels that need to be pushed to the top in the rewriting operation) for changing the cell state from s_i to s_j. Without loss of generality, let's assume here that $p_1 \geqslant p_2 \geqslant \cdots \geqslant p_{i-1} \geqslant p_{i+1} \geqslant \cdots \geqslant p_{q-1} \geqslant p_q$, and that $d(s_i,s_1) \geqslant d(s_i,s_2) \geqslant \cdots \geqslant d(s_i,s_{i-1}) \geqslant d(s_i,s_{i+1}) \geqslant \cdots \geqslant d(s_i,s_{q-1}) \geqslant d(s_i,s_q)$. To minimize the expected rewriting cost, the ideal solution is a code that decodes s_j as j for $j \in Q$. Denote by α the expected rewriting cost of this ideal solution. Next, we design a prefix code B with this property: $\forall j \in Q$, if $j \neq i$, its corresponding codeword length, y_j, is at most $3d(s_i,s_j)$; if $j = i$, then $y_j = 1$. We will prove that such a prefix code B exists. Next, let A be an optimal prefix code, and for $j \in Q$, let x_j denote the corresponding codeword length. Let β denote the expected rewriting cost of A for the next rewrite. By definition, $\sum_{1\leqslant j\leqslant q} p_j x_j \leqslant \sum_{1\leqslant j\leqslant q} p_j y_j$. Since $x_i \geqslant 1 = y_i$, $\beta \leqslant \sum_{1\leqslant j\leqslant q, j\neq i} p_j x_j \leqslant \sum_{1\leqslant j\leqslant q, j\neq i} p_j y_j \leqslant \sum_{1\leqslant j\leqslant q, j\neq i} 3p_j d(s_i,s_j) = 3\alpha$. So the expected rewriting cost of an optimal prefix code is at most three times that of an ideal solution.

We skip the proof of Lemma 14 due to its simplicity.

Lemma 14. *Let $a_1, a_2, \ldots, a_{n-1}$ be non-negative integers. There exists a prefix code with a_i codewords of length i for $i = 1,2,\ldots,n-1$ (i.e., the code has a_i leaves in layer i of the tree C) if and only if $\sum_{i=1}^{n-1} \frac{a_i}{n!/(n-i)!} \leqslant 1$.*

Let us define the following sequence of integers $b_1,\ldots,b_{n-1}$:

$$b_i = \begin{cases} 1 & i = 1 \\ \left|\mathcal{S}_{i/3}(u)\right| = \frac{n!}{(n-i/3)!} - \frac{n!}{(n+1-i/3)!} & 2 \leqslant i \leqslant n-2 \\ & i \equiv 0 \pmod 3 \\ 0 & 2 \leqslant i \leqslant n-2 \\ & i \not\equiv 0 \pmod 3 \\ \frac{n!}{2} - \sum_{i=1}^{n-2} b_i & i = n-1 \end{cases}$$

Lemma 15. *When $q = n!/2$, there exists a prefix code that has b_i codewords of length i, for $i = 1,2,\ldots,n-1$.*

Proof: We use an induction on n and the conclusion in Lemma 14. Let $f(n)$ denote $\sum_{i=1}^{n-1} \frac{b_i}{n!/(n-i)!}$. When $n = 2,3,4,5,6,7,8$, $f(n) = \frac{1}{2}, \frac{2}{3}, \frac{17}{24}, \frac{87}{120}, \frac{7}{10}, \frac{3377}{5040}, \frac{163}{252}$, respectively. So $f(n) \leqslant 1$ when $n \leqslant 8$. By Lemma 14, a prefix code exists when $n \leqslant 8$. This serves as the base case.

We now show that when $n \geqslant 8$, $f(n)$ monotonically decreases in n. By the definition of $b_1, b_2, \ldots, b_{n-1}$, we get:

- If $n = 3m$ for some integer $m \geqslant 1$, then $f(n) = \frac{1}{3m} + \sum_{i=1}^{m-1} \frac{1}{(3m-i+1)\prod_{j=i+1}^{3i-1}(3m-j)} + \sum_{i=m}^{3m-2} \frac{1}{(3m-i+1)\prod_{j=i+1}^{3m-2}(3m-j)}$.
- If $n = 3m+1$ for some integer $m \geqslant 1$, then $f(n) = \frac{1}{3m+1} + \sum_{i=1}^{m} \frac{1}{(3m+2-i)\prod_{j=i+1}^{3i-1}(3m+1-j)} + \sum_{i=m+1}^{3m-1} \frac{1}{(3m+2-i)\prod_{j=i+1}^{3m-1}(3m+1-j)}$.
- If $n = 3m+2$ for some integer $m \geqslant 1$, then $f(n) = \frac{1}{3m+2} + \sum_{i=1}^{m} \frac{1}{(3m+3-i)\prod_{j=i+1}^{3i-1}(3m+2-j)} + \sum_{i=m+1}^{3m} \frac{1}{(3m+3-i)\prod_{j=i+1}^{3m}(3m+2-j)}$.

So when $m \geqslant 2$, we get $f(3m+3) - f(3m+2) = \frac{1}{(2m+3)\cdot(2m+1)!} - \frac{1}{(3m+3)(3m+2)} + \sum_{i=1}^{m} \frac{(3m+3-i)(3m+3-3i)-(3m+4-i)(3m+2-i)}{\prod_{j=i-4}^{3i-3}(3m-j)} < \frac{1}{(2m+3)\cdot(2m+1)!} - \frac{1}{(3m+3)(3m+2)} < 0$, so $f(3m+3) < f(3m+2)$. Similarly, we get $f(3m+4) < f(3m+3)$ and $f(3m+5) < f(3m+4)$ when $m \geqslant 2$. So $f(n)$ monotonically decreases when $n \geqslant 8$ increases. Since $f(n) \leqslant 1$ when $n \leqslant 8$, $f(n) \leqslant 1$ for all n. So by Lemma 14, the conclusion holds. ∎

Let $q = n!/2$, and let $i \in Q$ be the stored value at this moment. Let B_i denote a prefix code that has b_j codewords of length j, for $j = 1, 2, \ldots, n-1$. Label the $q = n!/2$ codewords of B_i as $w_1, w_2, \ldots, w_q$ based on their codeword length; specifically, if $j < k$, then w_j is no longer than w_k. The codewords of B_i are mapped to the alphabet Q in the following way: w_1 represents the value i; for any $2 \leqslant j < k \leqslant q$, if w_j represents $a_j \in Q$ and w_k represents $a_k \in Q$, then $p_{a_j} \geqslant p_{a_k}$. Then, we have the following lemma.

Lemma 16. *let $i \in Q$ denote the current value of the stored data. For the next rewrite, the expected cost (i.e., the number of cell charge levels that need to be pushed to the top during the rewriting operation) for the code B_i is at most three times that of any rank modulation code.*

Proof: We first consider a generic rank modulation code. The state of the n cells before the rewrite is a vertex u in the *transition graph* $G = (V, E)$. (The definition of the transition graph is shown in the previous subsection.) The $q = n!/2$ vertices in G that are closest to u are the vertices in the ball $\mathcal{B}_{n-2}(u)$. Let us label those vertices as $v_1, v_2, \ldots, v_{n!/2}$ based on their distance to u; specifically, if $j < k$, then $d(u, v_j) \leqslant d(u, v_k)$. (So $v_1 = u$.) Among them, the number of vertices at distance j to u equals $\left|\mathcal{S}_j(u)\right|$, for $j = 0, 1, \ldots, n-2$.

Let $\pi_1, \pi_2, \ldots, \pi_q$ be a permutation of the alphabet Q, such that u represents the value $\pi_1 = i$, and the probabilities $p_{\pi_2} \geqslant p_{\pi_3} \geqslant \cdots \geqslant p_{\pi_q}$. Clearly, for any rank modulation code, the expected cost for the next rewrite must be greater than or equal to $\sum_{j=2}^{q} p_{\pi_j} d(u, v_j)$.

Let us use $\left|w_j\right|$ to denote the length of the codeword w_j. By the definition of $b_1, b_2, \ldots, b_q$, it is easy to verify that $\left|w_j\right| \leqslant 3d(u, v_j)$ for $j = 2, 3, \ldots, q$. Therefore, $\sum_{j=2}^{q} p_{\pi_j} \left|w_j\right| \leqslant 3 \sum_{j=2}^{q} p_{\pi_j} d(u, v_j)$.

Let us say that w_j represents the value $s_j \in Q$, for $j = 1, 2, \ldots, q$. Since $s_1 = i = \pi_1$ and $p_{s_2} \geqslant p_{s_3} \geqslant \cdots \geqslant p_{s_q}$, we get $p_{s_j} = p_{\pi_j}$ for $1 \leqslant j \leqslant q$. So $\sum_{j=2}^{q} p_{s_j} \left|w_j\right| = \sum_{j=2}^{q} p_{\pi_j} \left|w_j\right| \leqslant 3 \sum_{j=2}^{q} p_{\pi_j} d(u, v_j)$. When code B_i is used, the expected cost for the next rewrite is at most $\sum_{j=2}^{q} p_{s_j} \left|w_j\right|$. So the conclusion holds. ∎

The following theorem shows that when $q \leqslant n!/2$, the optimal prefix code (which is constructed by the algorithm in this section) is a 3-approximation of the optimal rank modulation code.

Theorem 17. *When $q \leqslant n!/2$, given the stored value at any moment (which can be anything in the alphabet Q), for the next rewrite, the expected cost (i.e., the number of cells charge levels that need to be pushed to the top during the rewriting operation) for an optimal prefix code is at most three times the expected cost for all rank modulation codes. Therefore, for any number of rewrites, the average rewrite cost of an optimal prefix code is at most three times that of any rank modulation code.*

Proof: Let C_{opt} denote an optimal prefix code (i.e., a prefix code that minimizes the weighted average codeword length). Let Q' be a new alphabet of $q' = n!/2$ numbers, whose associated rewrite probabilities are $p_1, p_2, \ldots, p_l, 0, 0, \ldots, 0$, respectively. Let C'_{opt} be an optimal prefix code for the new alphabet Q'. Clearly, the weighted average codeword length of C_{opt} is less than or equal to that of C'_{opt}, because the code C'_{opt} is more restricted. For $i = 1, 2, \ldots, q$, Let B_i be the same code as defined before, whose alphabet is also Q'. Clearly, among the three weighted average codeword lengths of C_{opt}, C'_{opt} and B_i, that of C_{opt} is the smallest and that of B_i is the largest. Therefore, if we use x_j (respectively, y_j^i)) to denote the length of the codeword that represents the value $j \in Q$ in code C_{opt} (respectively, code B_i), then $\sum_{j=1}^{q} p_j x_j \leqslant \sum_{j=1}^{q} p_j y_j^i$. By the definition of B_i, $y_i^i = 1$. Since $x_i \geqslant 1$, $\sum_{1 \leqslant j \leqslant l, j \neq i} p_j x_j \leqslant \sum_{1 \leqslant j \leqslant q, j \neq i} p_j y_j^i$.

Let us say that the stored value at this moment is i. The proof of Lemma 16 shows that for the next rewrite, $\sum_{1 \leqslant j \leqslant q, j \neq i} p_j y_j^i$ is at most three times the expected rewrite cost of any rank modulation code. For the next rewrite, the expected rewrite cost of C_{opt} is at most $\sum_{1 \leqslant j \leqslant q, j \neq i} p_j x_j$. The rest of the theorem follows naturally. ∎

We have shown that for an optimal prefix code, whose construction is presented in this paper, the rewrites increase the cells' highest charge level at a rate that is at most three times the optimal rate, when $q \leqslant n!/2$. With a similar analysis, we can prove the following result:

Theorem 18. *When $n \geqslant 4$ and $q \leqslant n!/6$, the average rewrite cost of an optimal prefix code is at most twice that of any rank modulation code.*

Proof: See Appendix. ∎

V. Conclusion

In this paper, we present a novel data storage scheme, *rank modulation*, for flash memories. We present several Gray code

constructions for rank modulation, as well as its data rewriting schemes. The presented coding schemes are optimized for cell programming cost in several different aspects.

APPENDIX

In this appendix, we prove Theorem 18. The general approach is similar to the way we have proved Theorem 17, so we only specify some details that are relatively important here.

We define a series of numbers $b_1, b_2, \ldots, b_{n-1}$ as follows. $b_1 = 1$. For $i = 2, 3, \ldots, n-2$, if i is a multiple of 2, then b_i equals the size of a sphere of radius $i/2$, which is $\frac{n!}{(n-i/2)!} - \frac{n!}{(n+1-i/2)!}$ by Lemma 9; otherwise, b_i equals 0. $b_{n-1} = \frac{n!}{6} - \sum_{i=1}^{n-2} b_i$. We now prove the existence of a specific prefix code.

Lemma 19. *When $n \geqslant 4$ and $q = n!/6$, there exists a prefix code that has b_i codewords of length i, for $i = 1, 2, \ldots, n-1$.*

Proof: We use an induction on n and the conclusion in Lemma 14. Let $f(n)$ denote $\sum_{i=1}^{n-1} \frac{b_i}{n(n-1)\cdots(n-i+1)}$. When $n = 4, 5, 6, 7, 8$, $f(n) = \frac{1}{2}, \frac{21}{40}, \frac{21}{40}, \frac{17}{35}, \frac{142}{315}$, respectively. So $f(n) \leqslant 1$ when $n \leqslant 8$. By Lemma 14, the prefix code exists when $n \leqslant 8$. This serves as the base case.

We now show that when $n \geqslant 8$, $f(n)$ monotonically decreases as n increases. By the definition of $b_1, b_2, \ldots, b_{n-1}$, we get:

- If $n = 2m$ for some integer $m \geqslant 4$, then $f(n) = \frac{1}{2m} + \sum_{i=1}^{m-1} \frac{1}{(2m-i+1)\prod_{j=i+1}^{2i-1}(2m-j)} + \sum_{i=m}^{2m-3} \frac{1}{(2m-i+1)\prod_{j=i+1}^{2m-2}(2m-j)}$.
- If $n = 2m+1$ for some integer $m \geqslant 4$, then $f(n) = \frac{1}{2m+1} + \sum_{i=1}^{m} \frac{1}{(2m+2-i)\prod_{j=i+1}^{2i-1}(2m+1-j)} + \sum_{i=m+1}^{2m-2} \frac{1}{(2m+2-i)\prod_{j=i+1}^{2m-1}(2m+1-j)}$.

So when $m \geqslant 4$, we get $f(2m+1) - f(2m) = \frac{1}{(m+2)\cdot m!} - \frac{1}{2m(2m+1)} + \sum_{i=1}^{m} \frac{(2m+1-i)(2m+1-2i)-(2m+2-i)(2m-i)}{\prod_{j=i-2}^{2i-1}(2m-j)} < \frac{1}{(m+2)\cdot m!} - \frac{1}{2m(2m+1)} < 0$, so $f(2m+1) < f(2m)$. Similarly, we get $f(2m+2) < f(2m+1)$. So $f(n)$ monotonically decreases when $n \geqslant 8$ increases. Since $f(n) \leqslant 1$ when $n \leqslant 8$, $f(n) \leqslant 1$ for all n. So by Lemma 14, the conclusion holds. ∎

We skip the rest of the proof because it is very similar to the 3-approximation case.

REFERENCES

[1] A. Bandyopadhyay, G. J. Serrano, and P. Hasler, "Programming analog computational memory elements to 0.2% accuracy over 3.5 decades using a predictive method," in *Proceedings of the IEEE International Symposium on Circuits and Systems*, 2005, pp. 2148–2151.

[2] V. Bohossian, A. Jiang, and J. Bruck, "Buffer coding for asymmetric multi-level memory," in *Proc. IEEE ISIT*, 2007.

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

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

[5] T. Etzion, "Personal communication," Oct. 2007.

[6] F. Gray, "Pulse code communication," U.S. Patent 2632058, March 1953.

[7] M. Grossi, M. Lanzoni, and B. Riccò, "Program schemes for multilevel flash memories," *Proceedings of the IEEE*, vol. 91, no. 4, pp. 594–601, 2003.

[8] F. K. Hwang, "Generalized huffman trees," *SIAM Journal Appl. Math.*, vol. 37, no. 1, pp. 124–127, 1979.

[9] A. Jiang, "On the generalization of error-correcting WOM codes," in *Proc. IEEE ISIT*, 2007.

[10] A. Jiang, V. Bohossian, and J. Bruck, "Floating codes for joint information storage in write asymmetric memories," in *Proc. IEEE ISIT*, 2007.

[11] A. Jiang, M. Schwartz, and J. Bruck, "Error-correcting codes for rank modulation," 2008, in preparation.

[12] H. Nobukata et al., "A 144-Mb, eight-level NAND flash memory with optimized pulsewidth programming," *IEEE J. Solid-State Circuits*, vol. 35, no. 5, pp. 682–690, 2000.

[13] C. D. Savage, "A survey of combinatorial Gray codes," *SIAM Rev.*, vol. 39, no. 4, pp. 605–629, 1997.

[14] R. Sedgewick, "Permutation generation methods," *Computing Surveys*, vol. 9, no. 2, pp. 137–164, 1977.