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, current flash memories 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 [3]. 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 [9], [5], [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.

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

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* [2], 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 exploit the full representational power of rank modulation, and data rewriting schemes. Error-correcting codes for rank modulation are presented in a companion paper [8].

The original Gray code [4] has been generalized in countless ways, and has been used in a wide range of applications. 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 [10], and the more specific [11] (and references therein). In [11] 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 present a balanced construction, which is a new permutation generation algorithm, that optimizes the transition step size and is suitable for block deflation.

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 [6]. 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.

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 $s_1, s_2, \dots, s_m$ 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 $\{1, 2, \dots, n\}$. An ordered set of n flash memory cells named $1, 2, \dots, n$, each containing a distinct charge level, induces a permutation of $[n]$ by writing the cell names in descending charge level $[a_1, a_2, \dots, a_n]$, 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.

We consider the basic minimal-cost operation on a given state to be a "push-to-the-top", by which a single cell has its charge level increased to become the highest of the set. Thus, the set T of minimal-cost transitions between states consists of $n-1$ functions $t_i, 2 \leqslant i \leqslant n$:
$t_i([a_1, \dots, a_{i-1}, \boldsymbol{a_i}, a_{i+1}, \dots, a_n]) = [\boldsymbol{a_i}, a_1, \dots, a_{i-1}, a_{i+1}, \dots, a_n]$.

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 cyclic and complete 3-RMGC is given below. The permutations are the columns being read from left to right. The sequence of transitions is: $t_2, t_3, t_3, t_2, t_3, t_3$.*

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

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 $t^{(1)}, t^{(2)}, \dots, t^{((n-1)!)}$ 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 $[1, 2, \dots, n]$, and then use the transitions $t^{(1)}, t^{(2)}, \dots, t^{((n-1)!-1)}$ 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 $[2, 1, 3, \dots, n-1, n]$.

We now use t_n to create the first permutation of the second block, and then use $t^{(1)}, t^{(2)}, \dots, t^{((n-1)!-1)}$ again to create the entire second block. We repeat this process $n-1$ times, i.e., use the sequence of transitions $t^{(1)}, t^{(2)}, \dots, t^{((n-1)!-1)}, t_n$ 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 $n, n-1, \dots, 3, 1$. 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 (where each box represents a single block, and $\rightsquigarrow$ denotes the sequence of transitions $t^{(1)}, \dots, t^{((n-1)!-1)}$):

$$\boxed{\begin{array}{ccc} 1 & & 2 \\ 2 & & 1 \\ \vdots & \rightsquigarrow & \vdots \\ n-1 & & n-1 \\ n & & n \end{array}} \xrightarrow{t_n} \boxed{\begin{array}{ccc} n & & 2 \\ 2 & & n \\ \vdots & \rightsquigarrow & \vdots \\ n-2 & & n-2 \\ n-1 & & n-1 \end{array}} \xrightarrow{t_n} \cdots \xrightarrow{t_n} \boxed{\begin{array}{ccc} 3 & & 2 \\ 2 & & 3 \\ \vdots & \rightsquigarrow & \vdots \\ n & & n \\ 1 & & 1 \end{array}}$$

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 $t^{(1)}, \dots, t^{((n-1)!)}$ such that its last transition is t_{n-1}[2]. For convenience we denote the rotated sequence by $\tau^{(1)}, \dots, \tau^{(n-1)!}$, where $\tau^{(n-1)!} = t_{n-1}$. The first permutation in the block is $[a_1, a_2, \dots, a_{n-1}, 2]$, and the last one is $[a_2, \dots, a_{n-1}, a_1, 2]$. In C' we find a transition of the following form: $[a_2, \dots, a_{n-1}, 2, a_1] \xrightarrow{t_{n-1}} [2, a_2, \dots, a_{n-1}, a_1]$. 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 bellow, where it is easy to see that 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

[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.

[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.

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

$t^{(1)}, \dots, t^{((n-1)!-1)}$ contains at least one occurrence of t_2, and is replicated $n-1$ times, $n \geqslant 3$. Therefore, we conclude that:

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

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, $c_i(p) = \max_k \{c_k(p-1)\} + 1$.

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 a complete n-RMGC is at least $n+1$, for $n \geqslant 3$.

We call an n-RMGC with jump cost $n+1$ a *balanced n-RMGC*. A balanced code is especially suitable if block deflation is possible (Section I). We show a construction that turns any $(n-1)$-RMGC into a balanced n-RMGC while retaining properties such as being cyclic or complete. The resulting recursive scheme is a new permutation generation algorithm, that obeys the geometric constraints of flash memories.

Theorem 5. *Given a cyclic and complete $(n-1)$-RMGC C_{n-1}, defined by the transitions $t_{i_1}, \dots, t_{i_{(n-1)!}}$, then the following transitions define an n-RMGC, denoted by $\mathcal{C}_n$, that is cyclic, complete and* balanced*:*

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

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*: $\overrightarrow{t_i}([a_1, \dots, a_{n-i}, \mathbf{a_{n-i+1}}, a_{n-i+2}, \dots, a_n]) = [a_1, \dots, a_{n-i}, a_{n-i+2}, \dots, a_n, \mathbf{a_{n-i+1}}]$.

Because C_{n-1} is cyclic and complete, using $\overrightarrow{t_{i_1}}, \dots, \overrightarrow{t_{i_{(n-1)!}}}$ starting with $[a_1, \dots, a_{n-1}]$ produces a complete cycle through S_{n-1}, and using them starting with $[a_1, \dots, a_n]$ creates a cycle through all the $(n-1)!$ permutations of $[n]$ where a_1 is fixed on the first position.

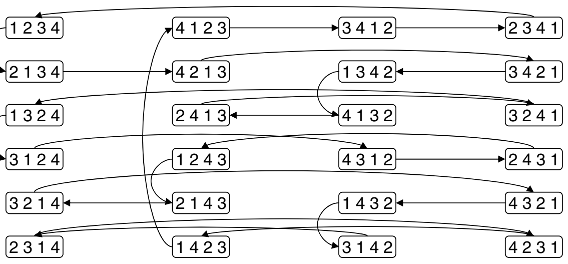

Figure 1. Recursive construction of the balanced 4-RMGC.

The $(n-1)!$ permutations of $[n]$ produced by $\overrightarrow{t_{i_1}}, \dots, \overrightarrow{t_{i_{(n-1)!}}}$ are also representatives of the $(n-1)!$ distinct orbits of the permutations of $[n]$ under the operation t_n. This 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:

$$\underbrace{t_{n-i_1+1}, \overbrace{t_n, \dots, t_n}^{n-1 \text{ times}}}_{\overrightarrow{t_{i_1}}}, \underbrace{t_{n-i_2+1}, \overbrace{t_n, \dots, t_n}^{n-1 \text{ times}}}_{\overrightarrow{t_{i_2}}}, \dots, \underbrace{t_{n-i_{(n-1)!}+1}, \overbrace{t_n, \dots, t_n}^{n-1 \text{ times}}}_{\overrightarrow{t_{i_{(n-1)!}}}}.$$

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 $\mathcal{C}_n$ is also cyclic and complete. ■

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

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

Example 7. *Fig. 1 shows the recursive balanced 4-RMGC. The permutations are represented as 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.*

The recursive balanced n-RMGC of Theorem 5 is optimal with respect to the following asymptotic measure. In practice, it is important to optimize the cost of deciding the transition that generates the next permutation. We define a *step* to be a single query of the form "what is the i-th highest charged cell?". If we start with $[a_1,\dots,a_n]$, then a fraction of $\frac{n-1}{n}$ of the transitions are t_n, and they occur whenever the cell a_1 is not the highest charged one. Of the cases where a_1 is highest charged, by recursion, a fraction $\frac{n-2}{n-1}$ of the transitions are determined by just one more query, and so on. At the basis of the recursion, permutations over two elements require zero additional 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.

IV. Rewriting with Rank Modulation Codes

In this section, we study rewriting data using the rank modulation scheme. The objective is to minimize the expensive cell erasure operations, which, in turn, requires us to maximize the number of times the data can be modified before a cell erasure becomes necessary (i.e., when the highest cell charge level reaches the highest allowed value). 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 internal representation alphabet size. In our case, data is stored internally in one of $n!$ different permutations. Let us assume the user alphabet is $Q = \{1,2,\dots,q\}$. 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 for* 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. In the following sections, we first present a decoding scheme that strictly optimizes the rewriting cost for the worst case. Then, we extend the construction to optimize the average rewriting cost 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 $u, v \in V$, define the directed distance $d(u,v)$ as the number of edges in the shortest directed path from u to v. Clearly, $0 \leqslant d(u,v) \leqslant n-1$. 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, $\mathcal{B}_r(u) = \bigcup_{0\leqslant i\leqslant r} \mathcal{S}_r(u)$. We skip the proof of Lemma 8 due to space constraint. Interested readers please see [7].

Lemma 8. *$\forall\, u \in V$ and $0 \leqslant r \leqslant n-1$, $|\mathcal{B}_r(u)| = \frac{n!}{(n-r)!}$. For $1 \leqslant r \leqslant n-1$, $|\mathcal{S}_r(u)| = \frac{n!}{(n-r)!} - \frac{n!}{(n-r+1)!}$.*

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. (Please see [7] for proof.)

Lemma 9. *For any decoding scheme and any current cell state, there exists $\alpha \in Q$ such that the cost of a rewriting operation for α is at least ρ. (ρ is as defined above.)*

Next, we present an optimal code construction.

Construction 10. *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 11. *Let $n = 3$ and $q = 3$. Since $|\mathcal{B}_1(u)| = 3$, $\rho = 1$. 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. Thus we have:

Theorem 12. *The coding scheme in Construction 10 is optimal in terms of minimizing the worst-case rewriting cost.*

B. Optimizing the Average Cost of Rewriting

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, the input symbol is drawn i.i.d. from $Q = \{1,2,\dots,q\}$ 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.

we present below a prefix code that is optimal in terms of its own design objective. Furthermore, we will prove that when $q \leqslant n!/2$, the prefix code is a 3-approximation of any optimal rank-modulation solution. We will also show that when $n \geqslant 4$ and $q \leqslant n!/6$, the prefix code is a 2-approximation.

The prefix code consists of q codewords of variable lengths, which represent the q values in Q. Each codeword is a prefix of a permutation from S_n. No codeword is allowed to be the prefix of another codeword. Let $a = [a_1, a_2, \dots, a_i]$ 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 where the labels on the $n!$ paths from its root to leaves are all the $n!$ permutations. An example is shown in Fig. 2(a). The vertices are in n layers, with the root in layer 0 and leaves in layer n. A prefix code corresponds to a subtree C of T (see Fig. 2(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.

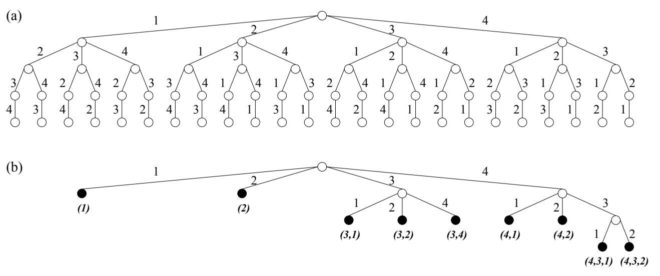

Figure 2. Prefix rank modulation code for $n = 4$ and $q = 9$. (a) The full permutation tree T. (b) A prefix code represented by a subtree C of T. The leaves represent the codewords, which are the labels beside the leaves.

For $i \in Q$, let c_i denote the codeword representing i, and let $|c_i|$ denote its length. (The codewords in Fig. 2(b) have minimum length of 1 and maximum length of 3.) Our objective is to *minimize the average codeword length*, $\sum_{i=1}^{q} p_i\,|c_i|$, which upper bounds the expected cost of each rewrite.

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

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 $\mathsf{opt}_i(\ell, m)$. 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: (1) $\mathsf{opt}_{n-1}(\ell, m) = (n-1)\sum_{k=1}^{\ell} p_k$ for $m \geqslant \ell > 0$; (2) $\mathsf{opt}_i(0, m) = 0$ for $i > 1$; (3) $\mathsf{opt}_i(\ell, m) = \min_{0 \leqslant j \leqslant \min\{\ell, m\}}\{\mathsf{opt}_{i+1}(\ell - j, \min\{q, (m-j)(n-i)\}) + \sum_{k=\ell-j+1}^{\ell} i p_k\}$ for $i < n-1, \ell > 0, m > 0$. 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 computes $\mathsf{opt}_{n-1}(\ell, m)$, $\mathsf{opt}_{n-2}(\ell, m)$, $\cdots$, $\mathsf{opt}_1(q, n)$ using the above recursions. Given their values, it is straightforward to determine in the optimal code, how many codewords are in each layer, and therefore determine the optimal code itself. For rewriting based on the code, to change the stored value to $i \in Q$, we simply push the $|c_i|$ cells in its codeword c_i to the top sequentially.

Theorem 13. *When $q \leqslant n!/2$, for every rewrite, the expected rewriting cost of an optimal prefix code is at most three times that of any rank modulation code. When $n \geqslant 4$ and $q \leqslant n!/6$, this ratio is at most two.*

Proof: We present the *sketch of the proof* for the case $q \leqslant n!/2$. (The case $n \geqslant 4$ and $q \leqslant n!/6$ can be analyzed similarly.) For the detailed proof, please refer to [7]. Let $i \in Q$ (resp., $s_i \in S_n$) denote the stored data (resp., cell state) at a given moment. Let $s_1, \cdots, s_{i-1}, s_{i+1}, \cdots, s_q$ denote the $q-1$ cell states whose distance from s_i in the *transition graph*, $d(s_i, s_j)$, are the smallest ones. WLOG, assume that $p_1 \geqslant \cdots \geqslant p_{i-1} \geqslant p_{i+1} \geqslant \cdots \geqslant p_q$, and that $d(s_i, s_1) \leqslant \cdots \leqslant d(s_i, s_{i-1}) \leqslant d(s_i, s_{i+1}) \leqslant \cdots \leqslant d(s_i, s_q)$. 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 $y_j = 1$. It can be proved 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. By definition, $\sum_{1 \leqslant j \leqslant q} p_j x_j \leqslant \sum_{1 \leqslant j \leqslant q} p_j y_j$. Since $x_i \geqslant 1 = y_i$, $\beta \leqslant \sum_{1 \leqslant j \leqslant q, j \neq i} p_j x_j \leqslant \sum_{1 \leqslant j \leqslant q, j \neq i} p_j y_j \leqslant \sum_{1 \leqslant j \leqslant q, j \neq i} 3 p_j d(s_i, s_j) = 3\alpha$. So the expected rewriting cost of an optimal prefix code is at most three times that of an ideal rank modulation code. ■

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.

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] 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.

[3] 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.

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

[5] 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.

[6] A. Jiang, V. Bohossian, and J. Bruck, "Floating codes for joint information storage in write asymmetric memories," in *Proc. IEEE ISIT*, 2007, pp. 1166–1170.

[7] A. Jiang, R. Mateescu, M. Schwartz, and J. Bruck, "Rank modulation for flash memories," California Institute of Technology, Tech. Rep., 2008. [Online]. Available: http://www.paradise.caltech.edu/etr.html

[8] A. Jiang, M. Schwartz, and J. Bruck, "Error-correcting codes for rank modulation," in *Proc. IEEE ISIT*, 2008.

[9] 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.

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

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