Information Representation and Coding for Flash Memories

Anxiao (Andrew) Jiang
Computer Science and Engineering Dept.
Texas A&M University
College Station, TX 77843, U.S.A.
Email: ajiang@cse.tamu.edu

Jehoshua Bruck
Electrical Engineering Department
California Institute of Technology
Pasadena, CA 91125, U.S.A.
Email: bruck@caltech.edu

Abstract—**Flash memories are a very widely used type of non-volatile memory. Like magnetic recording and optical recording, flash memories have their own distinct properties. These distinct properties introduce very interesting information-representation and coding problems, which address many aspects of a successful storage system. In this paper, we survey recent results in this area. A focus is placed on rewriting codes and rank modulation.**

I. Introduction

In this paper, we survey the recent results on information representation and coding for flash memories. Flash memories are a milestone in the development of the data storage technology. The applications of flash memories have expanded widely in recent years, and flash memories have become the dominating member in the family of non-volatile memories. Compared to magnetic recording and optical recording, flash memories are more suitable for many mobile-, embedded- and mass-storage applications. The reasons include their high speed, physical robustness, and easy integration with circuits.

The representation of data plays a key role in storage systems. Like magnetic recording and optical recording, flash memories have their own distinct properties, including block erasure, iterative cell programming, etc. [3] These distinct properties introduce very interesting information-representation and coding problems that address many aspects of a successful storage system, such as efficient data modification, error correction, etc. In this paper, we first introduce the flash memory model, then study some newly developed codes, including codes for rewriting data and the rank modulation scheme. We also survey the results on related topics in this area. A main theme for many of the topics is understanding how to store information in a medium that has asymmetric properties when it transits between different states.

II. Modelling Flash Memories

The basic storage unit in a flash memory is a floating-gate transistor [3]. We also call it a cell. Charge (e.g., electrons) can be injected into the cell using the hot-electron injection mechanism or the Fowler-Nordheim tunnelling mechanism, and the injected charge is trapped in the cell. (Specifically, the charge is stored in the floating-gate layer of the transistor.) The charge can also be removed from the cell using the Fowler-Nordheim tunnelling mechanism. The amount of charge in a cell determines its threshold voltage, which can be measured. The operation of injecting charge into a cell is called *writing* (or *programming*), removing charge is called *erasing*, and measuring the charge level is called *reading*. If we use two discrete charge levels to store data, the cell is called *single-level cell* (SLC) and can store one bit. If we use $q > 2$ discrete charge levels to store data, the cell is called *multi-level cell* (MLC) and can store $\log_2 q$ bits.

A prominent property of flash memories is *block erasure*. In a flash memory, cells are organized as blocks, each containing about 10^5 cells. While it is relatively easy to inject charge into a cell, to remove charge from any cell, the whole block containing it must be erased to the ground level (and then reprogrammed). This is called block erasure. The block erasure operation not only significantly reduces speed, but also reduces the lifetime of the flash memory [3]. This is because a block can only endure about $10^4 \sim 10^6$ erasures, after which the block may break down. Since the breaking down of a single block can make the whole memory stop working, it is important to balance the erasures performed to different blocks. This is called *wear leveling*. A commonly used wear-leveling technique is to balance erasures by moving data among the blocks, especially when the data are revised [10].

There are two main types of flash memories: NOR flash and NAND flash. A NOR flash memory allows random access to its cells. A NAND flash partitions every block into multiple sections called pages, and a page is the unit of a read or write operation. Compared to NOR flash, NAND flash may be much more restrictive on how its pages can be programmed, such as allowing a page to be programmed only a few times before erasure [10]. However, NAND flash enjoys the advantage of higher cell density.

The programming of cells is a noisy process. When charge is injected into a cell, the actual amount of injection is randomly distributed around the aimed value. An important thing to avoid during programming is overshooting, because to lower a cell's level, erasure is needed. A commonly used approach to avoid overshooting is to program a cell using multiple rounds of charge injection. In each round, a conservative amount of charge is injected into the cell. Then the cell's charge level (which we shall call *cell level*) is measured before the next round begins. With this approach, the charge level can gradually approach the target value and the programming precision is improved. The corresponding cost is the slowing down in the writing speed.

After cells are programmed, the data are not necessarily error-proof, because the cell levels can be changed by various errors over time. Some important error sources include *write disturb* and *read disturb* (disturbs caused by writing or reading), as well as leakage of charge from the cells (called the *retention problem*) [3]. The changes in the cell levels often have an asymmetric distribution in the up and the down directions, and the errors in different cells can be correlated.

In summary, flash memory is a storage medium with asymmetric properties. It is easy to increase a cell's level, but very costly to decrease it due to block erasure. The NAND flash may have more restrictions on reading and writing compared to NOR flash. The cell programming uses multiple rounds of charge injection to shift the cell level monotonically up toward the target value, to avoid overshooting and improve the precision. The cell levels can change over time due to various disturb mechanisms and the retention problem, and the errors can be asymmetric or correlated.

III. Codes for Rewriting Data

In this section, we discuss coding schemes for rewriting data in flash memories. The interest in this problem comes from the fact that if data are stored in the straightforward way, even to change one bit in the data, we may need to lower some cell's level, which would lead to the costly block erasure operation. It is interesting to see if there exist codes that allow data to be rewritten many times without block erasure. The flash memory model we use in this section is the *Write Asymmetric Memory (WAM)* model [15].

Definition 1. Write Asymmetric Memory (WAM)

In a write asymmetric memory, there are n cells. Every cell has $q \geq 2$ levels: levels $0, 1, \cdots, q-1$. The level of a cell can only increase, not decrease.

The Write Asymmetric Memory models the monotonic change of flash memory cells before the erasure operation. It is a special case of the *generalized write-once memory (WOM)* model, which allows the state transitions of cells to be any acyclic directed graph [6], [8], [27].

A. Floating Codes

Floating code generalizes the definition of WOM code by jointly storing multiple variables. The study of WOM code was started by Rivest and Shamir in their celebrated paper [27], where a single variable is stored. Jointly storing multiple variables can increase the number of supported rewrites, when each rewrite updates only one variable (or, not all variables). Let us first present the definition of floating codes [15].

Definition 2. Floating Code

We store k variables of alphabet size ℓ in a write asymmetric memory (WAM) with n cells of q levels. Every rewrite changes one of the k variables. Let $(c_1, \cdots, c_n) \in \{0, 1, \cdots, q-1\}^n$ denote the state of the memory (i.e., the levels of the n cells). Let $(v_1, \cdots, v_k) \in \{0, 1, \cdots, \ell-1\}^k$ denote the data (i.e., the values of the k variables). For any two memory states $(c_1, \cdots, c_n)$ and $(c'_1, \cdots, c'_n)$, we say $(c_1, \cdots, c_n) \geq (c'_1, \cdots, c'_n)$ if $c_i \geq c'_i$ for $i = 1, \cdots, n$.

A floating code has a decoding function F_d and an update function F_u. The decoding function $F_d : \{0, 1, \cdots, q-1\}^n \to \{0, 1, \cdots, \ell-1\}^k$ maps a memory state $s \in \{0, 1, \cdots, q-1\}^n$ to the stored data $F_d(s) \in \{0, 1, \cdots, \ell-1\}^k$. The update function (which represents a rewrite operation), $F_u : \{0, 1, \cdots, q-1\}^n \times \{1, 2, \cdots, k\} \times \{0, 1, \cdots, \ell-1\} \to \{0, 1, \cdots, q-1\}^n$, is defined as follows: "if the current memory state is s and the rewrite changes the i-th variable to value $j \in \{0, 1, \cdots, \ell-1\}$, then the rewrite operation will change the memory state to $F_u(s, i, j)$ such that $F_d(F_u(s, i, j))$ is the new data with the i-th variable changed to the value j." Naturally, since the memory is a write asymmetric memory, we require that $F_u(s, i, j) \geq s$.

Let t denote the number of rewrites (including the first write) guaranteed by the code. A floating code that maximizes t is called optimal.

Example 3.*[15] Here is a floating-code example for two binary variables. We store two binary variables in a WAM with $n = 3$ cells of q levels. Every rewrite changes the value of one variable. The floating code is shown in Fig. 1. In the figure, the numbers in a circle represent the memory state, the numbers beside a circle represent the two variables, and the arrows represent the transitions of the memory state.*

With every rewrite, the memory state moves up by one layer in the figure. For example, if $q \geq 3$ and a sequence of rewrites change the data as $(0,0) \to (1,0) \to (1,1) \to (0,1) \to (1,1) \to \cdots$, the memory state changes as $(0,0,0) \to (1,0,0) \to (1,0,1) \to (1,0,2) \to (1,1,2) \to \cdots$. The code in the figure has a periodic structure, where every period contains $2n - 1 = 5$ layers (as shown in the figure) and has the same topological structure. From one period to the next, the only difference is that the data $(0,0)$ is switched with $(1,0)$, and the data $(1,1)$ is switched with $(0,1)$. Given the finite value of q, we just need to truncate the graph up to the cell level $q-1$.

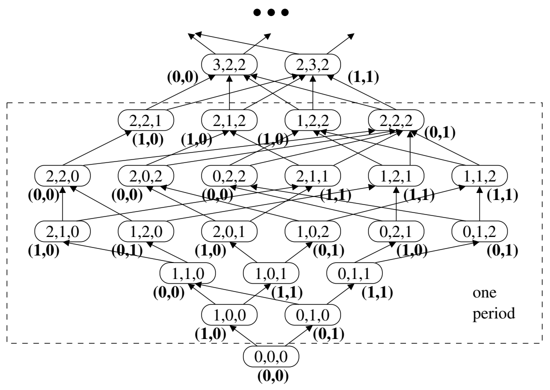


Fig. 1. An optimal floating code for $k = \ell = 2$, $n = 3$ and arbitrary q.

The floating code in the above example is generalized in [15] for any value of n and q (but still with $k = \ell = 2$), which is shown to guarantee $t = (n-1)(q-1) + \lfloor \frac{q-1}{2} \rfloor$ rewrites. It is optimal because it matches the following upper bound for t, which was proved in [15].

Theorem 4. *For any floating code, if $n \geq k(l-1) - 1$, then $t \leq [n - k(l-1) + 1] \cdot (q-1) + \lfloor \frac{[k(l-1)-1] \cdot (q-1)}{2} \rfloor$; if $n < k(l-1) - 1$, then $t \leq \lfloor \frac{n(q-1)}{2} \rfloor$.*

B. Generalized Rewriting Codes

We now extend floating codes to a more general definition of rewriting codes. We use a directed graph to represent how rewrites may change the stored data [18]. The other concepts – decoding function, update function, number of rewrites guaranteed by the rewriting code – are similar as before.

Definition 5. *[18]* GENERALIZED REWRITING
The stored data is represented by a directed graph $\mathcal{D} = (V_{\mathcal{D}}, E_{\mathcal{D}})$. The vertices $V_{\mathcal{D}}$ represent all the values that the data can take. There is a directed edge (u, v) from $u \in V_{\mathcal{D}}$ to $v \in V_{\mathcal{D}}$, $v \neq u$, iff a rewrite may change the data from value u to value v. The graph $\mathcal{D}$ is called the "data graph" *and the number of its vertices, corresponding to the input-alphabet size, is denoted by $L = |V_{\mathcal{D}}|$. Without loss of generality (w.l.o.g.), we assume the data graph to be strongly connected.*

It is simple to see that when the above notion is applied to floating codes, the alphabet size $L = \ell^k$, and the data graph $\mathcal{D}$ has constant in-degree and out-degree $k(\ell - 1)$. The out-degree of $\mathcal{D}$ shows by how much a rewrite can change data. It is an important parameter. In the following, we show a rewriting code for this generalized rewriting model. The code, called *Trajectory Code*, was proposed in [18].

Let $(c_1, \cdots, c_n) \in \{0, 1, \cdots, q-1\}^n$ denote the memory state. Let $V_{\mathcal{D}} = \{0, 1, \cdots, L-1\}$ denote the alphabet of the stored data. Let's present the trajectory code step by step, starting with its basic building blocks.

1) Linear Code and Extended Linear Code: We first look at a "Linear Code" for the case $n = L - 1$ and $q = 2$ [27].

Construction 6. *[27]* LINEAR CODE FOR $n = L - 1$, $q = 2$
The memory state $(c_1, \cdots, c_n)$ represents the data $\sum_{i=1}^{n} ic_i \mod (n+1)$. For every rewrite, change as few cells from level 0 to level 1 as possible to get the new data.

Example 7. *Let $n = 7$, $q = 2$ and $L = 8$. Using the linear code, the data represented by the memory state $(c_1, \cdots, c_7)$ is $\sum_{i=1}^{7} ic_i \mod 8$. If the rewrites change the data as $0 \rightarrow 3 \rightarrow 5 \rightarrow 2 \rightarrow 4$, the memory state can change as $(0,0,0,0,0,0,0) \rightarrow (0,0,1,0,0,0,0) \rightarrow (0,1,1,0,0,0,0) \rightarrow (0,1,1,0,1,0,0) \rightarrow (0,1,1,1,1,1,0)$.*

When $n \geq L$ and $q \geq 2$, we can generalize the Linear Code in the following way [18]. First, suppose $n = L$ and $q \geq 2$. We first use level 0 and level 1 to encode (as the Linear Code does), and let the memory state represent the data $\sum_{i=1}^{n} ic_i \mod n$. (Note that rewrites here will not change c_n.) When the code can no longer support rewriting, we increase all cell levels (including c_n) from 0 to 1, and start using cell levels 1 and 2 to store data in the same way as above, except that now, the data represented by the memory state $(c_1, \cdots, c_n)$ uses the formula $\sum_{i=1}^{n} i(c_i - 1) \mod n$. This process is repeated $q-1$ times in total. The general decoding function is therefore $\sum_{i=1}^{n} i(c_i - c_n) \mod n$.

Now we extend the above code to $n \geq L$ cells. We divide the n cells into $b = \lfloor n/L \rfloor$ groups of size L (some cells may remain unused), and sequentially apply the above code to the first group of L cells, then to the second group, and so on. We call this code the *Extended Linear Code*. It has been shown that the Linear Code guarantees $\frac{n+1}{4} + 1$ rewrites [27], while the Extended Linear Code guarantees $n(q-1)/8 = \Theta(nq)$ rewrites [18]. Both are asymptotically optimal in n.

2) Code for Large Alphabet Size L: We now consider the case where L is larger than n. The rewriting code we present here will reduce it to the case $n = L$ studied above.

Construction 8. *[18]* REWRITING CODE FOR $n < L \leq 2^{\sqrt{n}}$
Let b be the smallest positive integer value that satisfies $\lfloor n/b \rfloor^b \geq L$. For $i = 1, 2, \ldots, b$, let v_i be a symbol from an alphabet of size $\lfloor n/b \rfloor \geq L^{1/b}$. We may represent any symbol $v \in \{0, 1, \cdots, L-1\}$ as a vector of symbols $(v_1, v_2, \ldots, v_b)$. Partition the n flash cells into b groups, each with $\lfloor n/b \rfloor$ cells (some cells may remain unused). Encoding the symbol v into n cells is equivalent to the encoding of each v_i into the corresponding group of $\lfloor n/b \rfloor$ cells. As the alphabet size of each v_i equals the number of cells it is to be encoded into, we can use the Extended Linear Code to store v_i.

The above code construction guarantees $\frac{n(q-1)\log n}{16 \log L} = \Theta(\frac{nq \log n}{\log L})$ rewrites when $16 \leq n \leq L \leq 2^{\sqrt{n}}$ [18]. It is asymptotically optimal in the sense that when $n < L - 1$ and the data graph $\mathcal{D}$ is a complete graph, a rewriting code can guarantee at most $O(\frac{nq \log n}{\log L})$ rewrites. It should be noted that the above three codes –linear code, extended linear code, and Construction 8 – do not specify any constraint for the data graph $\mathcal{D}$, so $\mathcal{D}$ can be a complete graph.

3) Full Construction of Trajectory Code: Let's now consider more restricted rewrite operations. In many applications, a rewrite often changes only a (small) part of the data. So let's consider the case where a rewrite can change the data to at most Δ new values. This is the same as saying that in the data graph $\mathcal{D}$, the maximum out-degree is Δ. We call such graph $\mathcal{D}$ a "Bounded Out-degree Data Graph."

The code to be shown is given the name *Trajectory Code* in [18]. Its idea is to record the path in $\mathcal{D}$ along which the data changes, up to a certain length. When Δ is small, this approach is particularly helpful, because recording which outgoing edge a rewrite takes (one of Δ choices) is more efficient than recording the new data value (one of $L > \Delta$ choices).

We first outline the construction of the Trajectory Code. Its parameters will be specified soon.

Construction 9.[18] (OUTLINE OF TRAJECTORY CODE)

Let $n_0, n_1, n_2, \ldots, n_d$ be $d+1$ positive integers such that $\sum_{i=0}^{d} n_i = n$ is the number of cells. We partition the n cells into $d+1$ groups, each with $n_0, n_1, \ldots, n_d$ cells, respectively. We call them registers $S_0, S_1, \ldots, S_d$.

The encoding uses the following basic scheme: we start by using register S_0, called the anchor, *to record the value of the initial data $v_0 \in \{0, 1, \cdots, L-1\}$. For the next d rewrite operations we use a differential scheme: denote by $v_1, \ldots, v_d \in \{0, 1, \cdots, L-1\}$ the next d values of the rewritten data. In the i-th rewrite, $1 \leq i \leq d$, we store in register S_i the identity of the edge $(v_{i-1}, v_i) \in E_{\mathcal{D}}$. ($E_{\mathcal{D}}$ and $V_{\mathcal{D}}$ are the edge set and vertex set of the data graph $\mathcal{D}$, respectively.) We do not require a unique label for all edges globally, but rather require that* locally, *for each vertex in $V_{\mathcal{D}}$, its out-going edges have unique labels from $\{1, \cdots, \Delta\}$, where Δ denotes the maximal out-degree in the data graph $\mathcal{D}$.*

Intuitively, the first d rewrites are achieved by encoding the trajectory *taken by the input data sequence starting with the anchor data. After d such rewrites, we repeat the process by rewriting the next input from $\{0, 1, \cdots, L-1\}$ in the anchor S_0, and then continuing with d edge labels in $S_1, \cdots, S_d$.*

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

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

Note that each register S_i, for $i = 0, \ldots, d$, can be seen as a smaller rewriting code *whose data graph is* a complete graph *of either L vertices (for S_0) or Δ vertices (for $S_1, \ldots, S_d$). We use either the Extended Linear Code or the code of Construction 8 for rewriting in the $d+1$ registers.*

The parameters of the *Trajectory Code* are shown by the following construction. We assume that $n \leq L \leq 2^{\sqrt{n}}$.

Construction 10. TRAJECTORY CODE FOR $n \leq L \leq 2^{\sqrt{n}}$

If $\Delta \leq \lfloor \frac{n \log n}{2 \log L} \rfloor$, let $d = \lfloor \log L / \log n \rfloor = \Theta(\log L / \log n)$. If $\lfloor \frac{n \log n}{2 \log L} \rfloor \leq \Delta \leq L$, let $d = \lfloor \log L / \log \Delta \rfloor = \Theta(\log L / \log \Delta)$. In both cases, set the register sizes to $n_0 = \lfloor n/2 \rfloor$ and $n_i = \lfloor n/(2d) \rfloor$ for $i = 1, \ldots d$.

If $\Delta \leq \lfloor \frac{n \log n}{2 \log L} \rfloor$, apply the code of Construction 8 to register S_0, and apply the Extended Linear Code to registers $S_1, \cdots, S_d$. If $\lfloor \frac{n \log n}{2 \log L} \rfloor \leq \Delta \leq L$, apply the code of Construction 8 to the $d+1$ registers $S_0, \cdots, S_d$.

It is shown in [18] that when $\Delta \leq \lfloor \frac{n \log n}{2 \log L} \rfloor$, the trajectory code guarantees $\Theta(nq)$ rewrites; when $\lfloor \frac{n \log n}{2 \log L} \rfloor \leq \Delta \leq L$, the trajectory code guarantees $\Theta\left(\frac{nq \log n}{\log \Delta}\right)$ rewrites. Both are asymptotically optimal.

IV. RANK MODULATION

Rank modulation is a new data representation scheme. It was proposed in [20], [23] for two objectives: to eliminate the risk of over-injection of charge when programming cells, and to tolerate asymmetric errors better. Rank modulation uses the relative order of the cell levels, instead the absolute values of cell levels, to represent data. Let's look at a simple example. Let $[n]$ denote the set of integers $\{1, 2, \cdots, n\}$.

Example 11. *We partition the cells into groups of three cells each. Denote the three cells in a group by cell 1, cell 2, and cell 3. We use a permutation of $\{1, 2, 3\}$ – $[a_1, a_2, a_3]$ – to represent the relative order of the three cell levels as follows: cell a_1 has the highest level, and cell a_3 has the lowest level. (The cell levels considered in this section are real numbers. So no two cells can practically have the same level.)*

The three cells in a group can introduce six possible permutations: $[1,2,3], [1,3,2], [2,1,3], [2,3,1], [3,1,2], [3,2,1]$. So they can store up to $\log_2 6$ bits of information. To write a permutation, we program the cells from the lowest level to the highest level. For example, if the permutation to write is $[2,3,1]$, we first program cell 3 to make its level higher than that of cell 1, then program cell 2 to make its level higher than that of cell 3. This way, there is no risk of overshooting.

We use n to denote the number of cells in a group. As in the example, we use a permutation of $[n]$ – $[a_1, a_2, \cdots, a_n]$ – to denote the relative order of the cell levels such that cell a_1 has the highest level and cell a_n has the lowest level.

A. Rewriting Codes for Rank Modulation

Assume that the only operation we allow for rewriting data is the "push-to-top" operation: injecting charge into a cell to make its level higher than all the other cell levels in the same cell group. (The operation has no risk of overshooting.) Then, how to design good rewriting codes for rank modulation?

Let ℓ denote the alphabet size of the data symbol stored in a group of n cells. (Here $\ell \leq n!$.) Let the alphabet of the data symbol be $[\ell] = \{1, 2, \cdots, \ell\}$. Assume that a rewrite can change the data to any value in $[\ell]$. A rewriting code decodes every permutation $[a_1, a_2, \cdots, a_n]$ as a symbol in $[\ell]$. We define the cost of a rewrite as the number of "push-to-top" operations needed to change the permutation. It is observed that this rewrite cost is closely related to the "prefixes" of permutations [20]. Let's see an example.

Example 12. *Let $n = 4$, $\ell = 5$. For $i = 1, \cdots, n$, we say $[a_1, \cdots, a_i]$ is a prefix of $[a_1, \cdots, a_n]$. (For example, $[1,2,3,4]$ and $[1,2,4,3]$ have the same prefix $[1,2]$.) Let $P([a_1, \cdots, a_i])$ denote all the permutations of $[n]$ with prefix*

$[a_1, \cdots, a_i]$. *By* $P([a_1, \cdots, a_i]) \mapsto j$, *we mean that the rewriting code decodes all the permutations with prefix* $[a_1, \cdots, a_i]$ *as symbol* $j \in [\ell]$. *Then, let's define a rewriting code as follows: "*$P([1]) \mapsto 1$, $P([2]) \mapsto 2$, $P([3]) \mapsto 3$, $P([4,1]) \mapsto 4$, $P([4,2]) \mapsto 5$.*" Since the maximum prefix length used in the code is two, the cost of rewriting is at most two.*

Let $\rho_{n,\ell}$ denote the smallest integer such that $\frac{n!}{(n-\rho_{n,\ell})!} \geq \ell$. It is shown in [20] that regardless of the current cell state, there is a always a "next rewrite" of cost at least $\rho_{n,\ell}$. It is also shown that there is a prefix-free code (like the one in the above example) of maximum prefix length $\rho_{n,\ell}$ [20]. (So it is worst-case optimal.) Prefix-free codes for minimum average rewriting cost were also studied in [20].

B. Error-correcting Codes for Rank Modulation

An error-correcting rank-modulation code is a subset of permutations that are far from each other based on some distance measure. In [23], the distance between two permutations A and B, $d(A, B)$, is defined as the minimum number of *adjacent transpositions* needed to change A into B (or B into A). Note that given a permutation, an *adjacent transposition* is the local exchange of two adjacent elements in the permutation: $[a_1, \ldots, a_{i-1}, a_i, a_{i+1}, a_{i+2}, \ldots, a_n]$ is changed to $[a_1, \ldots, a_{i-1}, a_{i+1}, a_i, a_{i+2}, \ldots, a_n]$. (For example, $d([2,1,3,4], [2,3,4,1]) = 2$ because we have adjacent transpositions: $[2,1,3,4] \rightarrow [2,3,1,4] \rightarrow [2,3,4,1]$.)

Define the adjacency graph of permutations, $G = (V, E)$, as follows. The graph G has $|V| = n!$ vertices, which represent the $n!$ permutations. Two vertices $u, v \in V$ are adjacent if and only if $d(u, v) = 1$. G is a regular undirected graph with degree $n-1$ and diameter $\frac{n(n-1)}{2}$. It is shown in [23] that G is a subgraph of the $2 \times 3 \times \cdots \times n$ linear array. Examples for $n = 3, 4$ are shown in Fig. 2. The following construction shows how to embed the permutation $[a_1, a_2, \cdots, a_n]$ as the vertex of coordinate $(x_1, x_2, \cdots, x_{n-1})$ in the $2 \times 3 \times \cdots \times n$ linear array (here $0 \leq x_i \leq i$): "*For* $i = 1, \cdots, n-1$, *in the permutation* $[a_1, a_2, \cdots, a_n]$, *the element* $i+1$ *is in front of* x_i *elements of the set* $\{1, 2, \cdots, i\}$." [23]

From the above embedding result, it immediately follows that if we construct an error-correcting code in the $2 \times 3 \times \cdots \times n$ linear array of minimum L_1-distance d, we also get an error-correcting code of rank modulation of minimum distance at least d. An one-error-correcting rank-modulation code was constructed using this method in [23], whose cardinality was proved to be at least half of optimal.

V. Extended Information Theoretic Results

Flash memory is a type of constrained memory. There has been a history of distinguished theoretical study on constrained memories. It includes the original work by Kuznetsov and Tsybakov on coding for defective memories [24]. Further developments on defective memories include [11], [13]. The write once memory (WOM) [27], write unidirectional memory (WUM) [26], [28], [30], and write efficient memory [1], [9] are also special instances of constrained memories. Among them, WOM is the most related to the Write-Asymmetric Memory model studied in this paper.

Write once memory (WOM) was defined by Rivest and Shamir in their original work [27]. In a WOM, a cell's state can change from 0 to 1 but not from 1 to 0. This model was later generalized with more cell states in [6], [8]. The objective of WOM codes is to maximize the number of times that the stored data can be rewritten. A number of very interesting WOM code constructions have been presented over the years, including the tabular codes, linear codes, etc. in [27], the linear codes in [6], the codes constructed using projective geometries [25], and the coset coding in [5]. Fine results on the capacity of WOM have been presented in [8], [12], [27], [31]. Furthermore, error-correcting WOM codes have been studied in [33]. In all the above works, the rewriting model assumes no constraints on the data, namely, the data graph $\mathcal{D}$ is a complete graph.

With the increasing importance of flash memories, new topics on coding for flash memories have been studied in recent years. For efficient data rewriting, floating codes and buffer codes were proposed in [15] and [2]. A floating code jointly encodes multiple variables, and every rewrite changes one variable. A buffer code, on the other hand, records the most recent data of a data stream. More results on floating codes that optimize the worst-case rewriting performance were presented in [16], [32]. Floating codes that also correct errors were studied in [14]. In [18], the rewriting problem was generalized based on the data graph model, and trajectory codes for complete data graphs and bounded-degree data graphs were presented. The paper [18] also contains a summary and comparison of previous results on rewriting codes.

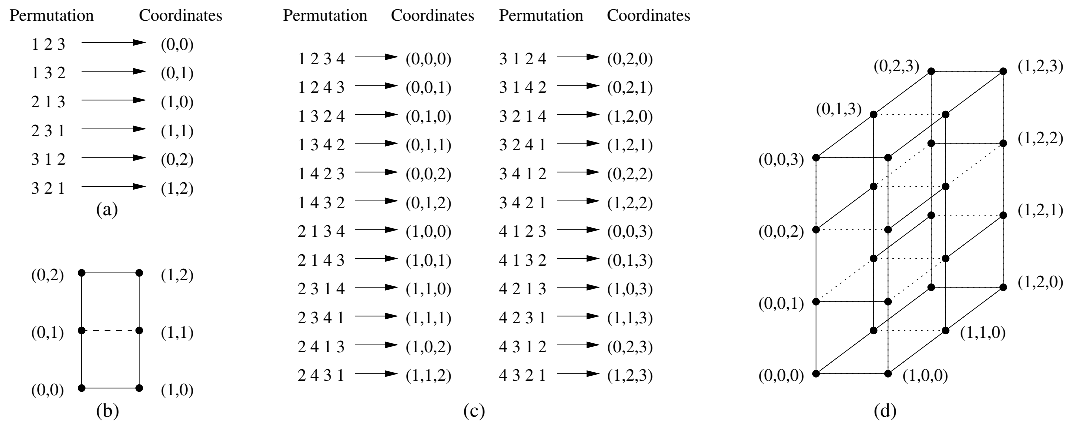
Optimizing rewriting codes for expected performance is also an interesting topic. In [7], floating codes of this type were designed based on Gray code constructions. In [18], randomized WOM codes of robust performance were proposed.

The rank modulation scheme was proposed and studied in [20], [21], [23]. In addition to rewriting [20] and error correction [23], a family of Gray codes for rank modulation were also presented [20], [21]. One application of the Gray codes is to map rank modulation to the conventional multi-level cells. A type of convolutional rank modulation codes, called *Bounded Rank Modulation*, was studied in [29].

To study the storage capacity of flash memories, it is necessary to understand how accruately flash cells can be programmed using the iterative and monotonic programming method. This was studied in [17] based on an abstract programming-error model.

The errors in flash cell levels often have an asymmetric property. In [4], error-correcting codes that correct asymmetric errors of limited magnitude were designed for flash memories.

In a storage system, to avoid the accumulation of errors, a common practice is to write the correct data back into the storage system once the errors accumulated in the data reach a certain threshold. This is called *memory scrubbing*. In flash memories, however, memory scrubbing is more difficult because to write one correct codeword back into the system, the whole block needs to be erased. A new type of error-



(a)

Permutation	Coordinates
1 2 3	(0,0)
1 3 2	(0,1)
2 1 3	(1,0)
2 3 1	(1,1)
3 1 2	(0,2)
3 2 1	(1,2)

(c)

Permutation	Coordinates	Permutation	Coordinates
1 2 3 4	(0,0,0)	3 1 2 4	(0,2,0)
1 2 4 3	(0,0,1)	3 1 4 2	(0,2,1)
1 3 2 4	(0,1,0)	3 2 1 4	(1,2,0)
1 3 4 2	(0,1,1)	3 2 4 1	(1,2,1)
1 4 2 3	(0,0,2)	3 4 1 2	(0,2,2)
1 4 3 2	(0,1,2)	3 4 2 1	(1,2,2)
2 1 3 4	(1,0,0)	4 1 2 3	(0,0,3)
2 1 4 3	(1,0,1)	4 1 3 2	(0,1,3)
2 3 1 4	(1,1,0)	4 2 1 3	(1,0,3)
2 3 4 1	(1,1,1)	4 2 3 1	(1,1,3)
2 4 1 3	(1,0,2)	4 3 1 2	(0,2,3)
2 4 3 1	(1,1,2)	4 3 2 1	(1,2,3)

Fig. 2. Coordinates of permutations, and embedding the adjacency graph of permutations, G, in the $2 \times 3 \times \cdots \times n$ array, L_n. In the two arrays, the solid lines are the edges in both G and L_n, and the dotted lines are the edges only in L_n. (a) Coordinates of permutations for $n = 3$. (b) Embedding G in L_n for $n = 3$. (c) Coordinates of permutations for $n = 4$. (d) Embedding G in L_n for $n = 4$.

correcting codes, called *Error-Scurbbing Codes*, were defined in [19] for multi-level cells. It is shown that even if the only allowed operation is to increase cell levels, a higher rate of ECC can still be achieved by actively scrubbing errors.

The block erasure property of flash memories affects not only rewriting and cell programming, but also data movement. In [22], it is shown that by appropriately using coding, the number of erasures needed for moving data among n NAND flash blocks can be reduced by a factor of $O(\log n)$.

Acknowledgment

This work was supported in part by the NSF CAREER Award CCF-0747415, the NSF grant ECCS-0802107, and the Caltech Lee Center for Advanced Networking.

References

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

[2] V. Bohossian, A. Jiang and J. Bruck, “Buffer codes for asymmetric multi-level memory,” in *Proc. ISIT*, 2007, pp. 1186-1190.

[3] P. Cappelletti, C. Golla, P. Olivo and E. Zanoni (*Ed.*), *Flash memories*, Kluwer Academic Publishers, 1st Edition, 1999.

[4] Y. Cassuto, M. Schwartz, V. Bohossian and J. Bruck, “Codes for multi-level flash memories: Correcting asymmetric limited-magnitude errors,” in *Proc. ISIT*, Nice, France, June 2007, pp. 1176-1180.

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

[6] A. Fiat and A. Shamir, “Generalized ‘write-once’ memories,” *IEEE Trans. on Inform. Theory*, vol. IT-30, no. 3, pp. 470–480, May 1984.

[7] H. Finucane, Z. Liu and M. Mitzenmacher, “Designing floating codes for expected performance,” in *Proc. 46th Annual Allerton Conference*, 2008.

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

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

[10] E. Gal and S. Toledo, “Algorithms and data structures for flash memories,” *ACM Computing Surveys*, vol. 37, no. 2, pp. 138-163, 2005.

[11] A. J. Han Vinck and A. V. Kuznetsov, “On the general defective channel with informed encoder and capacities of some constrained memories,” *IEEE Trans. on Inform. Theory*, vol. 40, no. 6, pp. 1866–1871, 1994.

[12] C. D. Heegard, “On the capacity of permanent memory,” *IEEE Trans. on Inform. Theory*, vol. IT-31, no. 1, pp. 34–42, Jan. 1985.

[13] C. Heegard and A. Gamal, “On the capacity of computer memory with defects,” *IEEE T. Inform. Theory*, vol. IT-29, no. 5, pp. 731–739, 1983.

[14] A. Jiang, “On the generalization of error-correcting WOM codes,” in *Proc. IEEE ISIT*, 2007, pp. 1391-1395.

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

[16] A. Jiang and J. Bruck, “Joint coding for flash memory storage,” in *Proc. IEEE ISIT*, 2008, pp. 1741-1745.

[17] A. Jiang and J. Bruck, “On the capacity of flash memories,” in *Proc. Int. Symp. Inform. Theory and Its Appl. (ISITA)*, 2008, pp. 94-99.

[18] A. Jiang, M. Langberg, M. Schwartz and J. Bruck, “Universal rewriting in constrained memories,” to appear in *Proc. IEEE ISIT*, June-July 2009.

[19] A. Jiang, H. Li and Y. Wang, “Error scrubbing codes for flash memories,” in *Proc. Canadian Workshop on Inform. Theory (CWIT)*, 2009, pp. 32-35.

[20] A. Jiang, R. Mateescu, M. Schwartz and J. Bruck, “Rank modulation for flash memories,” in *Proc. IEEE ISIT*, 2008, pp. 1731-1735.

[21] A. Jiang, R. Mateescu, M. Schwartz and J. Bruck, “Rank modulation for flash memories,” in *IEEE Trans. Information Theory*, vol. 55, no. 6, pp. 2659-2673, 2009.

[22] A. Jiang, R. Mateescu, E. Yaakobi, J. Bruck, P. Siegel, A. Vardy and J. Wolf, “Storage coding for wear leveling in flash memories,” to appear in *Proc. IEEE ISIT*, Seoul, Korea, June-July 2009.

[23] A. Jiang, M. Schwartz and J. Bruck, “Error-correcting codes for rank modulation,” in *Proc. IEEE ISIT*, 2008, pp. 1736-1740.

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

[25] F. Merkx, “WOM codes constructed with projective geometries,” *Traitment du Signal*, vol. 1, no. 2-2, pp. 227–231, 1984.

[26] W. M. C. J. van Overveld, “The four cases of write unidirectional memory codes over arbitrary alphabets,” *IEEE Trans. on Inform. Theory*, vol. 37, no. 3, pp. 872–878, 1991.

[27] R. L. Rivest and A. Shamir, “How to reuse a ‘write-once’ memory,” *Information and Control*, vol. 55, pp. 1-19, 1982.

[28] G. Simonyi, “On write-unidirectional memory codes,” *IEEE Trans. on Inform. Theory*, vol. 35, no. 3, pp. 663–667, May 1989.

[29] Z. Wang, A. Jiang and J. Bruck, “On the capacity of bounded rank modulation for flash memories,” to appear in *Proc. IEEE ISIT*, 2009.

[30] F. M. J. Willems and A. J. Vinck, “Repeated recording for an optical disk,” in *Proc. 7th Symp. Inform. Theory in the Benelux*, 1986, pp. 49-53.

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

[32] E. Yaakobi, A. Vardy, P. H. Siegel and J. K. Wolf, “Multidimensional flash codes,” in *Proc. 46th Annual Allerton Conference*, 2008.

[33] G. Zémor and G. Cohen, “Error-correcting WOM-codes,” *IEEE Trans. on Inform. Theory*, vol. 37, no. 3, pp. 730–734, May 1991.