4

Data Representation for Flash Memories

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

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

In this chapter, we introduce theories on data representation 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. These distinct properties introduce very interesting coding problems that address many aspects of a successful storage system, which include efficient data modification, error correction, and more. In this chapter, we first introduce the flash memory model, then study some newly developed codes, including codes for rewriting data and the rank modulation scheme. A main theme is understanding how to store information in a medium that has asymmetric properties when it transits between different states.

1. 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 into blocks. A typical block contains 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 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 charge level (which we shall call *cell 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.

2. 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 conventional 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 [16].

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, 29].
Let us first look at an inspiring example. The code can write two bits twice in only three single-level cells. It was proposed by Rivest and Shamir in their celebrated paper that started the study of WOM codes [29].
We always assume that before data are written into the cells, the cells are at level 0.

Example 2. *We store two bits in three single-level cells (i.e., $n = 3$ and $q = 2$). The code is shown in Fig. 1. In the figure, the three numbers in a circle represent the three cell levels, and the two numbers beside the circle represent the two bits. The arrows represent the transition of the cells. As we can see, every cell level can only increase.*
The code allows us to write the two bits at least twice. For example, if want to write "10", and later rewrite them as "01", we first elevate the cell levels to "0,1,0", then elevate them to "0,1,1".

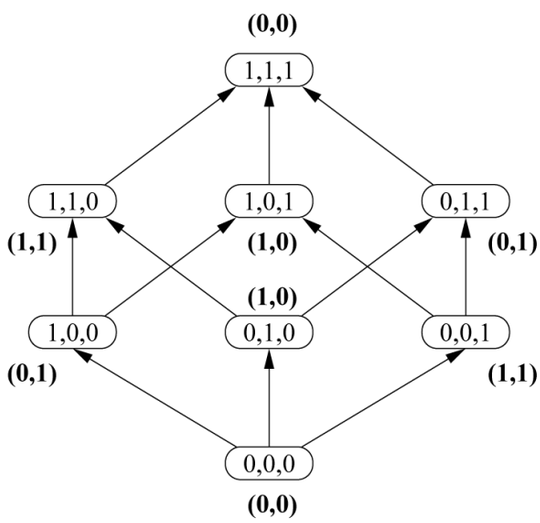

Fig. 1. Code for writing two bits twice in three single-level cells.

In the above example, a rewrite can completely change the data. In practice, often multiple data variables are stored by an application, and every rewrite changes only one of them. The joint coding of these data variables are useful for increasing the number of rewrites supported by coding. The rewriting codes in this setting have been named *Floating Codes* [16].

Definition 3. 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 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.*

The code in Fig. 1 is in fact a special case of the floating code, where the number of variables is only one. More specifically, the parameters for it is $k = 1, \ell = 4, n = 3, q = 2$ and $t = 2$.
Let us look at an example of floating codes for two binary variables.

Example 4. *We store two binary variables in a Write Asymmetric Memory with* n *cells of* q *levels. Every rewrite changes the value of one variable. The Floating codes for* $n = 1$*, 2 and 3 are shown in Fig.* 2*. As before, the numbers in a circle represent the memory state, the numbers beside a circle represent the data, and the arrows represent the transition of the memory state triggered by rewriting. With every rewrite, the memory state moves up by one layer in the figure. For example, if* $n = 3, q \geq 3$ *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 three codes in the figure all have a periodic structure, where every period contains $2n - 1$ *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* $q - 1$.

The floating codes in the above example are generalized in [16] for any value of n and q (but still with $k = \ell = 2$), and are shown to guarantee

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

rewrites. We now prove that this is optimal. First, we show an upper bound to t, the number of guaranteed rewrites, for floating codes [16].

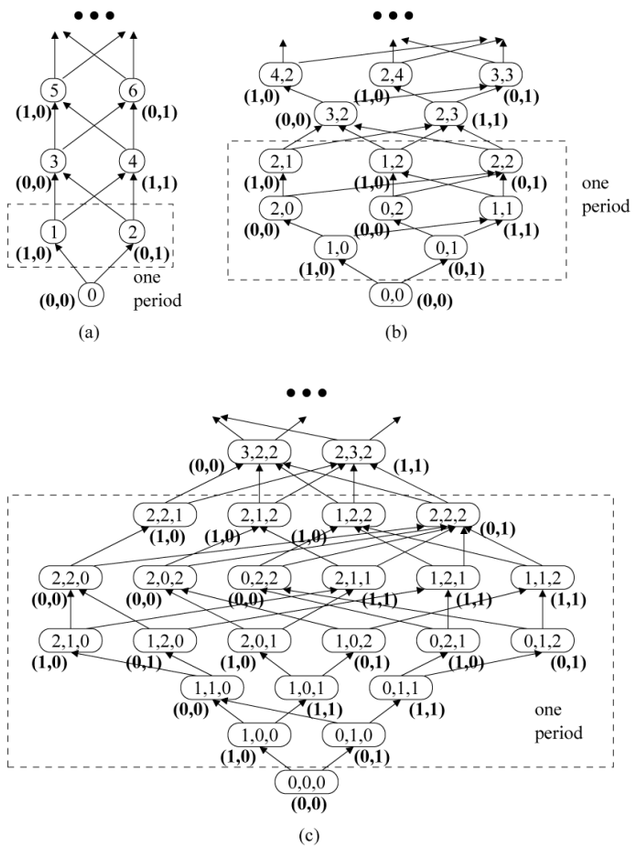

Fig. 2. Three examples of an optimal floating code for $k = 2, l = 2$ and arbitrary n, q. (a) $n = 1$. (b) $n = 2$. (c) $n = 3$.

Theorem 5. *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*

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

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

Consider a sequence of adversarial rewrites operations supported by a generic floating code. Suppose that x of those rewrite operations increase W_A by at least two, and suppose that y of them increase W_B by at least one. Since the maximum cell level is $q - 1$, we get $x \leq$

$\lfloor \frac{[k(l-1)-1]\cdot(q-1)}{2} \rfloor$ and $y \leq [n-k(l-1)+1]\cdot(q-1)$. So the number of rewrites supported by a floating code is at most $x+y \leq [n-k(l-1)+1]\cdot(q-1)+\lfloor \frac{[k(l-1)-1]\cdot(q-1)}{2} \rfloor$.
The case where $n < k(\ell-1)-1$ can be analyzed similarly. Call a rewrite operation "adversarial" if it increases $\sum_{i=1}^{n} c_i$ by at least two. It can be shown that there is always a adversarial choice for every rewrite, and any floating code can support at most $t \leq \lfloor \frac{n(q-1)}{2} \rfloor$ adversarial rewrites. □

When $k=\ell=2$, the above theorem gives the bound $t \leq (n-1)(q-1)+\lfloor \frac{q-1}{2} \rfloor$. It matches the number of rewrites guaranteed by the floating codes of Example 4 (and their generalization in [16]). So these codes are optimal.
Let us pause a little to consider the two given examples. In Example 2, two bits can be written twice into the three single-level cells. The total number of bits written into the memory is four (considering the whole history of rewriting), which is more than the number of bits the memory can store at any given time (which is three). In Example 4, every rewrite changes one of two binary variables and therefore reflects one bit of information. Since the code guarantees $(n-1)(q-1)+\lfloor \frac{q-1}{2} \rfloor$ rewrites, the total amount of information "recorded" by the memory (again, over the history of rewriting) is $(n-1)(q-1)+\lfloor \frac{q-1}{2} \rfloor \approx nq$ bits. In comparison, the number of bits the memory can store at any give time is only $n \log_2 q$.
Why can the total amount of information written into the memory (over the multiple rewrites) exceed $n \log_2 q$, the maximum number of bits the memory can store at any give time? It is because only the current value of the data needs to be remembered. Another way to understand it is that we are not using the cells sequentially. As an example, if there are n' single level cells and we increase one of their levels by one, there are n' choices, which in fact reflects $\log_2 n'$ bits of information (instead of one bit).
We now extend floating codes to a more general definition of rewriting codes. First, we use a directed graph to represent how rewrites may change the stored data. This definition was proposed in [19].

Definition 6. 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 rewriting operation may change the stored 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 $\mathcal{D}$ reveals how much change in the data a rewrite operation can cause. It is an important parameter. In the following, we show a rewriting code for this generalized rewriting model. The code, called *Trajectory Code*, was presented in [19].
Let $(c_1, \cdots, c_n) \in \{0,1,q-1\}^n$ 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.

Linear Code and Extended Linear Code

We first look at a *Linear Code* for the case $n=L-1$ and $q=2$. It was proposed by Rivest and Shamir in [29].

Definition 7. LINEAR CODE FOR $n = L - 1$ AND $q = 2$
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.

We give an example of the Linear Code.

Example 8. *Let* $n = 7$, $q = 2$ *and* $L = 8$. *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 $0 \to 3 \to 5 \to 2 \to 4$, *the memory state can change as* $(0,0,0,0,0,0,0) \to (0,0,1,0,0,0,0) \to (0,1,1,0,0,0,0) \to (0,1,1,0,1,0,0) \to (0,1,1,1,1,1,0)$.

The following theorem shows that the number of rewrites enabled by the Linear Code is asymptotically optimal in n, the number of cells. It was proved in [29].

Theorem 9. *The Linear Code guarantees at least* $\frac{n+1}{4} + 1$ *rewrites.*

Proof. We show that as long as at least $\frac{n+1}{2}$ cells are still of level 0, a rewrite will turn at most two cells from level 0 to level 1. Let $x \in \{0, 1, \cdots, n\}$ denote the current data value, and let $y \in \{0, 1, \cdots, n\}$ denote the new data value to be written, where $y \neq x$. Let z denote

$$y - x \mod (n+1).$$

If the cell c_z is of level 0, we can simply change it to level 1. Otherwise, let

$$S = \{i \in \{1, \cdots, n\} | c_i = 0\},$$

and let

$$T = \{z - s \mod (n+1) \mid s \in S\}.$$

Since $|S| = |T| \geq \frac{n+1}{2}$ and $|S \cup T| < n$ (zero and z are in neither set), the set $S \cap T$ must be nonempty. Their overlap indicates a solution to the equation

$$z = s_1 + s_2 \mod (n+1)$$

where s_1 and s_2 are elements of S. □

When $n \geq L$ and $q \geq 2$, we can generalize the Linear Code in the following way [19]. 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*.

Theorem 10. *Let $2 \leq L \leq n$. The Extended Linear Code guarantees $n(q-1)/8 = \Theta(nq)$ rewrites.*

Proof. The Extended Linear Code essentially consists of $(q-1)\lfloor \frac{n}{L} \rfloor \geq \frac{(q-1)n}{2L}$ Linear Codes. □

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 [19]. We start by assuming that $n < L \leq 2^{\sqrt{n}}$.

Construction 11. 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.

Theorem 12. *Let $16 \leq n \leq L \leq 2^{\sqrt{n}}$. The code in Construction 11 guarantees*

$$\frac{n(q-1)\log n}{16 \log L} = \Theta\left(\frac{nq \log n}{\log L}\right)$$

rewrites.

Proof. We first show that for b – the smallest positive integer value that satisfies $\lfloor n/b \rfloor^b \geq L$ – it holds that

$$b \leq \frac{2 \log L}{\log n}.$$

Note that for all $1 \leq x \leq \frac{\sqrt{n}}{2}$, we have $\lfloor n/x \rfloor^x \geq n^{x/2}$. Since $16 \leq n \leq L \leq 2^{\lfloor \sqrt{n} \rfloor}$, it is easy to verify that

$$\frac{2 \log L}{\log n} \leq \frac{\sqrt{n}}{2}.$$

Therefore,

$$\lfloor \frac{n \log n}{2 \log L} \rfloor^{\frac{2 \log L}{\log n}} \geq n^{\frac{\log L}{\log n}} = L,$$

which implies the upper bound for b.
Using Construction 11, the number of rewrites possible is bounded by the number of rewrites possible for each of the b cell groups. By Theorem 10 and the above upper bound for b, this is at least

$$\lfloor \frac{n}{b} \rfloor \cdot \frac{q-1}{8} \geq \left(\frac{n \log n}{2 \log L} - 1\right) \frac{q-1}{8} = \Theta\left(\frac{nq \log n}{\log L}\right).$$

□

For the codes we have shown so far, we have not mentioned any constraint on the data graph $\mathcal{D}$. Therefore, the above results hold for the case where the data graph is a complete graph. That is, a rewrite can change the data from any value to any other value. We now show that the above codes are asymptotically optimal. For the Linear Code and the Extended Linear Code, this is easy to see, because every rewrite needs to increase some cell level, so the number of rewrites cannot exceed $n(q-1) = O(nq)$. The following theorem shows that the rewriting code in Construction 11 is asymptotically optimal, too [19].

Theorem 13. *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.*

Proof. Let us consider some memory state s of the n flash cells, currently storing some value $v \in \{0, 1, \cdots, L-1\}$. The next rewrite can change the data into any of the other $L-1$ values. If we allow ourselves r operations of increasing a single cell level of the n flash cells (perhaps, operating on the same cell more than once), we may reach $\binom{n+r-1}{r}$ distinct new states. Let r be the maximum integer sastisfying $\binom{n+r-1}{r} < L-1$. So for the next rewrite, we need at least $r+1$ such operations in the worst case. Since we have a total of n cells with q levels each, the number of rewrite operations is upper bounded by

$$\frac{n(q-1)}{r+1} \leq \frac{n(q-1)}{\lfloor \frac{\log(L-1)}{1+\log n} \rfloor + 1} = O(\frac{nq\log n}{\log L}).$$

□

Trajectory Code Construction

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 [19]. 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. 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 rewrite operations 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 rewrite operations we would like to guarantee. 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 11 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 14. 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 size of the $d+1$ *registers 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* 11 *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* 11 *to the* $d+1$ *registers* $S_0, \cdots, S_d$.

The next three results show the asympotically optimality of the Trajectory Code (when Δ is small and large, respectively) [19].

Theorem 15. *Let* $\Delta \leq \lfloor \frac{n \log n}{2 \log L} \rfloor$. *The Trajectory Code of Construction* 14 *guarantees* $\Theta(nq)$ *rewrites.*

Proof. By Theorems 10 and 12, the number of rewrites possible in S_0 is equal (up to constant factors) to that of S_i ($i \geq 1$):

$$\Theta\left(\frac{n_0 q \log n_0}{\log L}\right) = \Theta\left(\frac{nq \log n}{\log L}\right) = \Theta\left(\frac{nq}{d}\right) = \Theta\left(n_i q\right)$$

Thus the total number of rewrites guaranted by the Trajectory Code is $d+1$ times the bound for each register S_i, which is $\Theta(nq)$. □

Theorem 16. *Let* $\lfloor \frac{n \log n}{2 \log L} \rfloor \leq \Delta \leq L$. *The Trajectory Code of Construction* 14 *guarantees* $\Theta\left(\frac{nq \log n}{\log \Delta}\right)$ *rewrites.*

Proof. By Theorem 12, the number of rewrites possible in S_0 is:

$$\Theta\left(\frac{n_0 q \log n_0}{\log L}\right) = \Theta\left(\frac{nq \log n}{\log L}\right)$$

Similarly the number of rewrites possible in S_i ($i \geq 1$) is:

$$\Theta\left(\frac{n_i q \log n_i}{\log \Delta}\right) = \Theta\left(\frac{nq \log n}{d \log \Delta}\right) = \Theta\left(\frac{nq \log n}{\log L}\right).$$

Here we use the fact that as $d \leq \log L$ it holds that $d = o(n)$ and $\log n_i = \Theta(\log n - \log d) = \Theta(\log n)$. Notice that the two expressions above are equal. Thus, as in Theorem 15, we conclude that the total number of rewrites guaranteed by the Trajectory Code is $d + 1$ times the bound for each register S_i, which is $\Theta\left(\frac{nq \log n}{\log \Delta}\right)$. □

The rewriting performance shown in the above theorem matches the bound shown in the following theorem. We omit its proof, which interested readers can find in [19].

Theorem 17. *Let $\Delta > \lfloor \frac{n \log n}{2 \log L} \rfloor$. There exist data graphs $\mathcal{D}$ of maximum out-degree Δ such that any rewriting code for $\mathcal{D}$ can guarantee at most $O\left(\frac{nq \log n}{\log \Delta}\right)$ rewrites.*

We have so far focused on rewriting codes with asymptotically optimal performance. It is interesting to study rewriting codes that are strictly optimal, like the floating code in Example 4. Some codes of this kind have been studied in [16, 17]. In some practical applications, more specific forms of rewriting can be defined, and better rewriting codes can be found. An example is the *Buffer Code* defined for streaming data [2]. Besides studying the worst-case performance of rewriting codes, the expected rewriting performance is equally interesting. Some rewriting codes for expected performance are reported in [7, 19].

3. Rank Modulation

We focus our attention now on a new data representation scheme called *Rank Modulation* [21, 23]. It uses the relative order of the cell levels, instead the absolute values of cell levels, to represent data. Let us first understand the motivations for the scheme.

Fast and accurate programming schemes for multi-level flash memories are a topic of significant research and design efforts. As mentioned before, the flash memory technology does not support charge removal from individual cells due to block erasure. As a result, an interative cell programming method is used. To program a cell, a sequence of charge injection operations are used to shift the cell level cautiously and monotinically toward the target charge level from below, in order to avoid undesired global erasures in case of overshoots. Consequently, the attempt to program a cell requires quite a few programming cycles, and it works only up to a moderate number of levels per cell.

In addition to the need for accurate programming, the move to more levels in cells also aggravates the reliability problem. Compared to single-level cells, the higher storage capacity of multi-level cells are obtained at the cost of a smaller gap between adjacent cell levels. Many of the errors in flash memories are asymmetric, meaning that they shift the cell levels more likely in one direction (up or down) than the other. Examples include the write disturbs, the read disturbs, and the charge leakage. It will be interesting to design coding schemes that tolerate asymmetric errors better.

The *Rank Modulation* scheme is therefore proposed in [21, 23], whose aim is to eliminate both the problem of overshooting while programming cells, and to tolerate asymmetric errors better. In this scheme, an ordered set of n cells stores the information in the permutation induced by the charge levels of the cells. In this way, no discrete levels are needed (i.e., no need for threshold levels) and only a basic charge-comparing operation (which is easy to implement) is required to read the permutation. If we further assume that the only programming operation allowed is raising the charge level of one of the cells above the current highest one (namely, *push-to-top*), then the overshoot problem is no longer relevant. Additionally, the technology may allow in the 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.
Let's look at a simple example of rank modulation. Let $[n]$ denote the set of integers $\{1, 2, \cdots, n\}$.

Example 18. *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* $[3] - [a_1, a_2, a_3]$ *– 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.*

In this section, 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.
Once a new data representation method is defined, tools of coding are required to make it useful. In this section, we focus on two tools: codes for rewriting, and codes for correcting errors.

3.1 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. Note that the push-to-top 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. 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 $[\ell]$. In general, ℓ might be smaller than $n!$, so we might end up having permutations that are not used. On the other hand, we can map several distinct permutations to the same symbol $i \in [\ell]$ in order to reduce the rewrite cost. We use S_n to denote the set of $n!$ permutations, and let $W_n \subseteq S_n$ denote the set of states (i.e., the set of permutations) that are used to represent information symbols. As before, for a rewriting code we can define two functions, a *decoding function*, F_d, and an *update function*, F_u.

Definition 19. *The decoding function $F_d : W_n \to [\ell]$ maps every state $s \in W_n$ to a value $F_d(s)$ in $[\ell]$. Given an "old state" $s \in W_n$ and a "new information symbol" $i \in [\ell]$, the update function $F_u : W_n \times [\ell] \to W_n$ produces a state $F_u(s,i)$ such that $F_d(F_u(s,i)) = i$.*

The more push-to-top operations are used for rewriting, the closer the highest cell level is to the maximum level possible. Once it reaches the maximum level possible, block erasure will be needed for the next rewrite. So we define the rewriting cost by measuring the number of push-to-top operations.

Definition 20. *Given two states $s_1, s_2 \in W_n$, the* cost *of changing s_1 into s_2, denoted $\alpha(s_1 \to s_2)$, is defined as the minimum number of "push-to-the-top" operations needed to change s_1 into s_2.*

For example, $\alpha([1,2,3] \to [2,1,3]) = 1$, $\alpha([1,2,3] \to [3,2,1]) = 2$.

Definition 21. *The* worst-case rewriting cost *of a code is defined as* $\max_{s \in W_n, i \in [\ell]} \alpha(s \to F_u(s,i))$. *A code that minimized this cost is called* optimal.

Before we present an optimal rewriting code, let's first present a lower bound on the worst-case rewriting cost. Define the *transition graph* $G = (V, E)$ as a directed graph with $V = S_n$, that is, with $n!$ vertices representing the permutations in S_n. For any $u, v \in V$, there is a directed edge from u to v iff $\alpha(u \to v) = 1$. G is a regular digraph, because every vertex has $n-1$ incoming edges and $n-1$ outgoing edges. The diameter of G is $\max_{u,v \in V} \alpha(u \to v) = n-1$.
Given a vertex $u \in V$ and an integer $r \in \{0, 1, \ldots, n-1\}$, define the *ball* centered at u with radius r as $\mathcal{B}_r^n(u) = \{v \in V \mid \alpha(u \to v) \leq r\}$, and define the *sphere* centered at u with radius r as $\mathcal{S}_r^n(u) = \{v \in V \mid \alpha(u \to v) = r\}$. Clearly,

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

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

Lemma 22. *For any $0 \leq r \leq n-1$,*

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

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

Proof. Fix a permutation $u \in V$. Let P_u be the set of permutations having the following property: for each permutation $v \in P_u$, the elements appearing in its last $n-r$ positions appear in the same relative order in u. For example, if $n = 5$, $r = 2$, $u = [1,2,3,4,5]$ and $v = [5,2,1,3,4]$, the last 3 elements of v – namely, $1, 3, 4$ – have the same relative order in u. It is easy to see that given u, when the elements occupying the first r positions in $v \in P_u$ are chosen, the last $n-r$ positions become fixed. There are $n(n-1)\cdots(n-r+1)$ choices for occupying the first r positions of $v \in P_u$, hence $|P_u| = \frac{n!}{(n-r)!}$. We will show that a vertex v is in $\mathcal{B}_r^n(u)$ if and only if $v \in P_u$.

Suppose $v \in \mathcal{B}_r^n(u)$. It follows that v can be obtained from u with at most r "push-to-the-top" operations. Those elements pushed to the top appear in the first r positions of v, so the last $n-r$ positions of v contain elements which have the same relative order in u, thus, $v \in P_u$. Now suppose $v \in P_u$. For $i \in [n]$, let v_i denote the element in the i-th position of v. One can transform u into v by sequentially pushing $v_r, v_{r-1}, \ldots, v_1$ to the top. Hence, $v \in \mathcal{B}_r^n(u)$. We conclude that $|\mathcal{B}_r^n(u)| = |P| = \frac{n!}{(n-r)!}$. Since $\mathcal{B}_r^n(u) = \bigcup_{0 \leq i \leq r} \mathcal{S}_i^n(u)$, the second claim follows. □

The following lemma shows a lower bound to the worst-case rewriting cost.

Lemma 23. *Fix integers n and ℓ, and define $\rho(n,\ell)$ to be the smallest integer such that $|\mathcal{B}^n_{\rho(n,\ell)}| \geq \ell$. For any code W_n and any state $s \in W_n$, there exists $i \in [\ell]$ such that $\alpha(s \to F_u(s,i)) \geq \rho(n,\ell)$, i.e., the worst-case rewriting cost of any code is at least $\rho(n,\ell)$.*

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

We now present a rewriting code construction. It will be shown that the code achieves the minimum worst-case rewriting cost. First, let us define the following notation.

Definition 24. *A* prefix sequence $\theta = [a^{(1)}, a^{(2)}, \ldots, a^{(m)}]$ *is a sequence of $m \leq n$ distinct symbols from $[n]$. The* prefix set $P_n(\theta) \subseteq S_n$ *is defined as all the permutations in S_n which start with the sequence θ.*

We are now in a position to construct the code.

Construction 25. REWRITING CODE FOR RANK MODULATION
Arbitrarily choose ℓ distinct prefix sequences, $\theta_1, \ldots, \theta_\ell$, each of length $\rho(n,\ell)$. Let us define $W_n = \bigcup_{i \in [\ell]} P_n(\theta_i)$ and map the states of $P_n(\theta_i)$ to i, i.e., for each $i \in [\ell]$ and $s \in P_n(\theta_i)$, set $F_d(s) = i$. Finally, to construct the update function F_u, given $s \in W_n$ and some $i \in [\ell]$, we do the following: let $[a_i^{(1)}, a_i^{(2)}, \ldots, a_i^{(\rho(n,\ell))}]$ be the first $\rho(n,\ell)$ elements which appear in all the permutations in $P_n(\theta_i)$. Apply push-to-top operations on the elements $a_i^{(\rho(n,\ell))}, \ldots, a_i^{(2)}, a_i^{(1)}$ in s to get a permutation $s' \in P_n(\theta_i)$ for which, clearly, $F_d(s') = i$. Set $F_u(s,i) = s'$.

Theorem 26. *The code in Construction* 25 *is optimal in terms of minimizing the worst-case rewriting cost.*

Proof. It is obvious from the description of F_u that the worst-case rewriting cost of the construction is at most $\rho(n,\ell)$. By Lemma 23 this is also the best we can hope for. □

Example 27. *Let $n = 3$, $\ell = 3$. Since $|\mathcal{B}_1^3| = 3$, it follows that $\rho(n,\ell) = 1$. We partition the $n! = 6$ states into $\frac{n!}{(n-\rho(n,\ell))!} = 3$ sets, which induce the mapping:*

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

The cost of any rewrite operation is 1.

If a probability distribution is known for the rewritten data, we can also define the performance of codes based on the average rewriting cost. This is studied in [21], where an variable-length prefix-free code is optimized. It is shown that the average rewriting cost of this prefix-free code is within a small constant approximation ratio of the minimum possible cost of all codes (prefix-free codes or not) under very mild conditions.

3.2 Error-correcting Codes for Rank Modulation

Error-correcting codes are often essential for the reliability of data. An error-correcting rank modulation code is a subset of permutations that are far from each other based on some distance measure. The distance between permutations needs to be defined appropriately according to the error model. In this section, we define distance as the number of adjacent transpositions. We emphasize, however, that this is not the only way to define distance. The results we present here were shown in [23].

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 $[a_1, \ldots, a_{i-1}, a_{i+1}, a_i, a_{i+2}, \ldots, a_n]$.

In the rank modulation model, the minimal change to a permutation caused by charge-level drift is a single adjacent transposition. Let us measure the number of errors by the minimum number of adjacent transpositions needed to change the permutation from its original value to its erroneous value. For example, if the errors change the permutation from $[2,1,3,4]$ to $[2,3,4,1]$, the number of errors is two, because at least two adjacent transpositions are needed to change one into the other: $[2,1,3,4] \rightarrow [2,3,1,4] \rightarrow [2,3,4,1]$.

For two permutations A and B, define their distance, $d(A,B)$, as the minimal number of adjacent transpositions needed to change A into B. This distance measure is called the *Kendall Tau Distance* in the statistics and machine-learning community [24], and it induces a metric over S_n. If $d(A,B) = 1$, A and B are called *adjacent*. Any two permutations of S_n are at distance at most $\frac{n(n-1)}{2}$ from each other. Two permutations of maximum distance are a reverse of each other.

Basic Properties

Theorem 28. *Let $A = [a_1, a_2, \ldots, a_n]$ and $B = [b_1, b_2, \ldots, b_n]$ be two permutations of length n. Suppose that $b_p = a_n$ for some $1 \leq p \leq n$. Let $A' = [a_1, a_2, \ldots, a_{n-1}]$ and $B' = [b_1, \ldots, b_{p-1}, b_{p+1}, \ldots, b_n]$. Then,*

$$d(A,B) = d(A',B') + n - p.$$

The above theorem can be proved by induction. It shows a recursive algorithm for computing the distance between two permutations. Let $A = [a_1, a_2, \ldots, a_n]$ and $B = [b_1, b_2, \ldots, b_n]$ be two permutations. For $1 \leq i \leq n$, let A_i denote $[a_1, a_2, \ldots, a_i]$, let B_i denote the subsequence of B that contains only those numbers in A_i, and let p_i denote the position of a_i in B_i. Then, since $d(A_1, B_1) = 0$ and $d(A_i, B_i) = d(A_{i-1}, B_{i-1}) + i - p_i$, for $i = 2, 3, \ldots, n$, we get

$$d(A,B) = d(A_n, B_n) = \frac{(n-1)(n+2)}{2} - \sum_{i=2}^{n} p_i.$$

Example 29. *Let* $A = [1,2,3,4]$ *and* $B = [4,2,3,1]$. *Then* $A_1 = [1]$, $A_2 = [1,2]$, $A_3 = [1,2,3]$, $A_4 = [1,2,3,4]$, $B_1 = [1]$, $B_2 = [2,1]$, $B_3 = [2,3,1]$, $B_4 = [4,2,3,1]$. *We get*

$$\begin{aligned} d(A_1,B_1) &= 0, \\ d(A_2,B_2) &= d(A_1,B_1) + 2 - p_2 = 0 + 2 - 1 = 1, \\ d(A_3,B_3) &= d(A_2,B_2) + 3 - p_3 = 1 + 3 - 2 = 2, \\ d(A_4,B_4) &= d(A_3,B_3) + 4 - p_4 = 2 + 4 - 1 = 5. \end{aligned}$$

Indeed, $d(A,B) = \frac{(n-1)(n+2)}{2} - \sum_{i=2}^{n} p_i = \frac{(4-1)(4+2)}{2} - (1+2+1) = 5$.

We now define a coordinate system for permutations. We fix $A = [1,2,\ldots,n]$. For every permutation $B = [b_1,b_2,\ldots,b_n]$, we define its *coordinates* as $X_B = (2-p_2, 3-p_3, \ldots, n-p_n)$. Here p_i is defined as above for $2 \le i \le n$. Clearly, if $X_B = (x_1,x_2,\ldots,x_{n-1})$, then $0 \le x_i \le i$ for $1 \le i \le n-1$.

Example 30. *Let* $A = [1,2,3,4,5]$. *Then* $X_A = (0,0,0,0)$. *If* $B = [3,4,2,1,5]$, *then* $X_B = (1,2,2,0)$. *If* $B = [5,4,3,2,1]$, *then* $X_B = (1,2,3,4)$. *The full set of coordinates for* $n = 3$ *and* $n = 4$ *are shown in Fig.* 3 *(a) and (c), respectively.* □

The coordinate system is equivalent to a form of Lehmer code (or Lucas-Lehmer code, inversion table) [26]. It is easy to see that two permutations are identical if and only if they have the same coordinates, and any vector $(y_1,y_2,\ldots,y_{n-1})$, $0 \le y_i \le i$ for $1 \le i \le n-1$, is the coordinates of some permutation in S_n. So there is a one-to-one mapping between the coordinates and the permutations.

Let $A \in S_n$ be a permutation. For any $0 \le r \le \frac{n(n-1)}{2}$, the set $\mathcal{B}_r(A) = \{B \in S_n \mid d(A,B) \le r\}$ is a *ball* of radius r centered at A. A simple relabeling argument suffices to show that the size of a ball does not depend on the choice of center. We use $|\mathcal{B}_r|$ to denote $|\mathcal{B}_r(A)|$ for any $A \in S_n$. We are interested in finding the value of $|\mathcal{B}_r|$. The following theorem presents a way to compute the size of a ball using polynomial multiplication.

Theorem 31. *For* $0 \le r \le \frac{n(n-1)}{2}$, *let* e_r *denote the coefficient of* x^r *in the polynomial* $\prod_{i=1}^{n-1} \frac{x^{i+1}-1}{x-1}$. *Then* $|\mathcal{B}_r| = \sum_{i=0}^{r} e_r$.

Proof. Let $A = [1,2,\ldots,n]$. Let $B = [b_1,b_2,\ldots,b_n]$ be a generic permutation. Let $X_B = (y_1,y_2,\ldots,y_{n-1})$ be the coordinates of B. By the definition of coordinates, we get $d(A,B) = \sum_{i=1}^{n-1} y_i$. The number of permutations at distance r from A equals the number of integer solutions to $\sum_{i=1}^{n-1} y_i = r$ such that $0 \le y_i \le i$. That is equal to the coefficient of x^r in the polynomial $\prod_{i=1}^{n-1}(x^i + x^{i-1} + \cdots + 1) = \prod_{i=1}^{n-1} \frac{x^{i+1}-1}{x-1}$. Thus, there are exactly e_r permutations at distance r from A, and $|\mathcal{B}_r| = \sum_{i=0}^{r} e_i$. □

Theorem 31 induces an upper bound for the sizes of error-correcting rank-modulation codes. By the sphere-packing principle, for such a code that can correct r errors, its size cannot exceed $n!/|\mathcal{B}_r|$.

Embedding of Permutation Adjacency Graph

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 $\frac{n(n-1)}{2}$. To study the topology of G, we begin with the follow theorem.

Lemma 32. *For two permutations $A = [a_1, a_2, \ldots, a_n]$ and $B = [b_1, b_2, \ldots, b_n]$, let their coordinates be $X_A = (x_1, x_2, \ldots, x_{n-1})$ and $X_B = (y_1, y_2, \ldots, y_{n-1})$. If A and B are adjacent, then $\sum_{i=1}^{n-1} |x_i - y_i| = 1$.*

The above lemma can be proved by induction. Interested readers can see [23] for details.
Let $L_n = (V_L, E_L)$ denote a $2 \times 3 \times \cdots \times n$ linear array graph. L_n has $n!$ vertices V_L. Each vertex is assigned integer coordinates $(x_1, x_2, \ldots, x_{n-1})$, where $0 \leq x_i \leq i$ for $1 \leq i \leq n-1$. The distance between vertices of L_n is the L_1 distance, and two vertices are adjacent (i.e., have an edge between them) if and only if their distance is one.
We now build a bijective map $P : V \to V_L$. Here V is the vertex set of the adjacency graph of permutations $G = (V, E)$. For any $u \in V$ and $v \in V_L$, $P(u) = v$ if and only if u, v have the same coordinates. By Lemma 32, if two permutations are adjacent, their coordinates are adjacent in L_n. So we get:

Theorem 33. *The adjacency graph of permutations is a subgraph of the $2 \times 3 \times \cdots \times n$ linear array.*

We show some examples of the embedding in Fig. 3. It can be seen that while each permutation has $n-1$ adjacent permutations, a vertex in the array can have a varied degree from $n-1$ to $2n-3$. Some edges of the array do not exist in the adjacency graph of permutations.

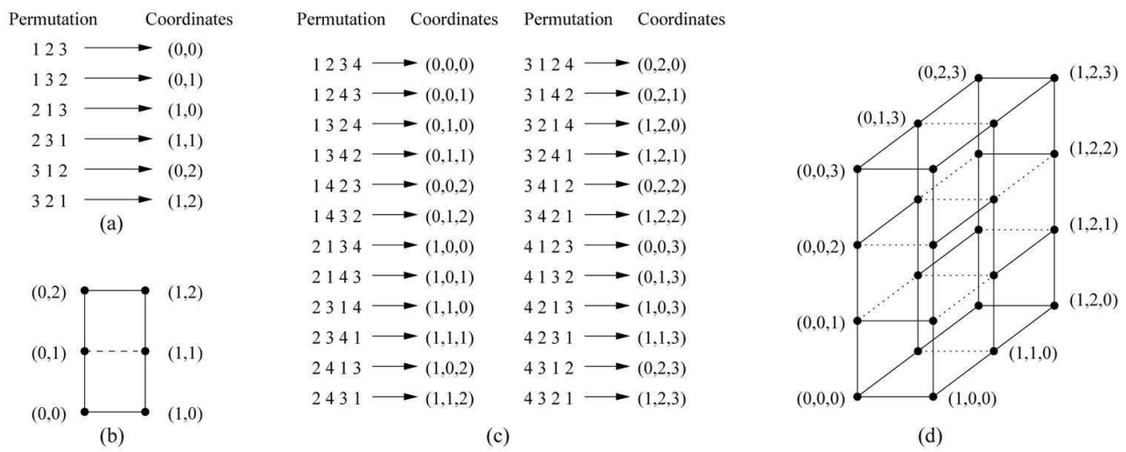

Fig. 3. Coordinates of permutations, and embedding the adjacency graph of permutations, G, in the $2 \times 3 \times \cdots \times n$ array, L_n. In the two arrays, the solid lines are the edges in both G and L_n, and the dotted lines are the edges only in L_n. (a) Coordinates of permutations for $n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

The observation that the permutations' adjacency graph is a subgraph of a linear array shows an approach to design error-correcting rank-modulation codes based on Lee-metric codes. We skip the proof of the following theorem due to its simplicity.

Theorem 34. *Let C be a Lee-metric error-correcting code of length $n-1$, alphabet size no less than n, and minimum distance d. Let C' be the subset of codewords of C that are contained in the array L_n. Then C' is an error-correcting rank-modulation code with minimum distance at least d.*

Single-error-correcting Rank-Modulation Code

We now present a family of rank-modulation codes that can correct one error. The code is based on the perfect sphere packing in the Lee-metric space [11]. The code construction is as follows.

Construction 35. (SINGLE-ERROR-CORRECTING RANK-MODULATION CODE)
Let C_1, C_2 denote two rank-modulation codes constructed as follows. Let A be a general permutation whose coordinates are $(x_1, x_2, \ldots, x_{n-1})$. Then A is a codeword in C_1 if and only if the following equation is satisfied:

$$\sum_{i=1}^{n-1} i x_i \equiv 0 \pmod{2n-1}.$$

A is a codeword in C_2 if and only if the following equation is satisfied:

$$\sum_{i=1}^{n-2} i x_i + (n-1) \cdot (-x_{n-1}) \equiv 0 \pmod{2n-1}.$$

Between C_1 and C_2, choose the code with more codewords as the final output. □

We analyze the code size of Construction 35.

Lemma 36. *The rank-modulation code built in Construction 35 has a minimum cardinality of $\frac{(n-1)!}{2}$.*

Proof. Let $H = (V_H, E_H)$ be a $2 \times 3 \times \cdots \times (n-1) \times (2n-1)$ linear array. Every vertex in H has integer coordinates $(x_1, x_2, \ldots, x_{n-1})$, where $0 \leq x_i \leq i$ for $1 \leq i \leq n-2$, and $-n+1 \leq x_{n-1} \leq n-1$.
Given any choice of $(x_1, x_2, \ldots, x_{n-2})$ of the coordinates, we would like to see if there is a solution to x_{n-1} (note that $-n+1 \leq x_{n-1} \leq n-1$) that satisfies the following equation:

$$\sum_{i=1}^{n-1} i x_i \equiv 0 \pmod{2n-1}.$$

Since $\sum_{i=1}^{n-1} i x_i = (n-1)x_{n-1} + \sum_{i=1}^{n-2} i x_i$, and $n-1$ and $2n-1$ are co-prime integers, there is exactly one solution to x_{n-1} that satisfies the above equation. If $x_{n-1} \geq 0$, clearly $(x_1, x_2, \ldots, x_{n-1})$ are the coordinates of a codeword in the code C_1. If $x_{n-1} \leq 0$, then $\sum_{i=1}^{n-2} i x_i + (n-1) \cdot [-(-x_{n-1})] \equiv 0 \pmod{2n-1}$, so $(x_1, x_2, \ldots, x_{n-2}, -x_{n-1})$ are the coordinates of a codeword in the code C_2.
Since $0 \leq x_i \leq i$ for $1 \leq i \leq n-2$, there are $(n-1)!$ ways to choose $x_1, x_2, \ldots, x_{n-2}$. Each choice generates a codeword that belongs either to C_1 or C_2. Therefore, at least one of C_1 and C_2 has cardinality no less than $\frac{(n-1)!}{2}$. □

Lemma 37. *The rank-modulation code built in Construction 35 can correct one error.*

Proof. It has been shown in [11] that for an infinite k-dimensional array, vertices whose coordinates $(x_1, x_2, \ldots, x_k)$ satisfy the condition $\sum_{i=1}^{k} i x_i \equiv 0 \pmod{2k+1}$ have a minimum L_1 distance of 3. Let $k = n-1$. Note that in Construction 35, the codewords of C_1 are a subset of the above vertices, while the codewords in C_2 are a subset of the mirrored image of the above vertices, where the last coordinate x_{n-1} is mapped to $-x_{n-1}$. Since the permutations' adjacency graph is a subgraph of the array, the minimum distance of C_1 and C_2 is at least 3. Hence, the code built in Construction 35 can correct one error. □

Theorem 38. *The code built in Construction* 35 *is a single-error-correcting rank-modulation code whose cardinality is at least half of optimal.*

Proof. Every permutation has $n-1$ adjacent permutations, so the size of a radius-1 ball, $|\mathcal{B}_1|$, is n. By the sphere packing bound, a single-error-correcting rank-modulation code can have at most $\frac{n!}{n} = (n-1)!$ codewords. The code in Construction 35 has at least $(n-1)!/2$ codewords. □

4. 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 [25]. Further developments on defective memories include [12, 14]. The write once memory (WOM) [29], write unidirectional memory (WUM) [28, 30, 32], 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 chapter.

Write once memory (WOM) was defined by Rivest and Shamir in their original work [29]. 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 [29], the linear codes in [6], the codes constructed using projective geometries [27], and the coset coding in [5]. Fine results on the capacity of WOM have been presented in [8, 13, 29, 33]. Furthermore, error-correcting WOM codes have been studied in [35]. 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 the efficient rewriting of data, floating codes and buffer codes were defined and studied in [16] and [2], respectively. 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 [17, 34]. Floating codes that also correct errors were studied in [15]. In [19], 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 [19] 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 [19], randomized WOM codes of robust performance were proposed.

The rank modulation scheme was proposed and studied in [21, 23]. In addition to rewriting [21] and error correction [23], a family of Gray codes for rank modulation were also presented [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 [31].

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 [18] 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-correcting codes, called *Error-Scurbbing Codes*, were defined in [20] 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 be still 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 $O(\log n)$.
The rewriting codes and the rank modulation scheme are useful not only for flash memories, but also for other constrained memories. A promonent example is the phase-change memory (PCM). In a phase-change memory, a memory cell can be switched between a crystalline state and an amorphous state. Intermediate states are also possible. Since changing the cell to the crystalline state takes a substantially longer time than changing it to the amorphous state, the memory is closely related to the Write-Asymmetric Memory model. How to program PCM cells with more intermediate states fast and reliably is an active ongoing topic. The rank modulation scheme may provide an effective tool in this area.

Acknowledgment

We would like to thank all our co-authors for their collaborative work in this area. In particular, we would like to thank Mike Langberg and Moshe Schwartz for many of the main results discussed in this chapter.

5. 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. IEEE International Symposium on Information Theory (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 multilevel flash memories: Correcting asymmetric limited-magnitude errors," in *Proc. IEEE International Symposium on Information Theory (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, Sep. 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, Jan. 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," in *ACM Computing Surveys*, vol. 37, no. 2, pp. 138-163, June 2005.
[11] S. W. Golomb and L. R. Welch, "Perfect codes in the Lee metric and the packing of polyominoes," *SIAM J. Appl. Math.*, vol. 18, no. 2, pp. 302–317, Jan. 1970.
[12] 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.
[13] C. D. Heegard, "On the capacity of permanent memory," *IEEE Trans. on Inform. Theory*, vol. IT-31, no. 1, pp. 34–42, Jan. 1985.
[14] C. D. Heegard and A. A. E. Gamal, "On the capacity of computer memory with defects," *IEEE Trans. on Inform. Theory*, vol. IT-29, no. 5, pp. 731–739, Sep. 1983.
[15] A. Jiang, "On the generalization of error-correcting WOM codes," in *Proc. IEEE International Symposium on Information Theory (ISIT'07)*, 2007, pp. 1391-1395.
[16] A. Jiang, V. Bohossian and J. Bruck, "Floating codes for joint information storage in write asymmetric memories," in *Proc. IEEE International Symposium on Information Theory (ISIT)*, 2007, pp. 1166-1170.
[17] A. Jiang and J. Bruck, "Joint coding for flash memory storage," in *Proc. IEEE International Symposium on Information Theory (ISIT)*, 2008, pp. 1741-1745.
[18] A. Jiang and J. Bruck, "On the capacity of flash memories," in *Proc. International Symposium on Information Theory and Its Applications (ISITA)*, 2008, pp. 94-99.
[19] A. Jiang, M. Langberg, M. Schwartz and J. Bruck, "Universal rewriting in constrained memories," in *Proc. IEEE International Symposium on Information Theory (ISIT)*, Seoul, Korea, June-July 2009. Technical report online at `http://www.paradise.caltech.edu/papers/etr096.pdf`.
[20] A. Jiang, H. Li and Y. Wang, "Error scrubbing codes for flash memories," in *Proc. Canadian Workshop on Information Theory (CWIT)*, May 2009, pp. 32-35.
[21] A. Jiang, R. Mateescu, M. Schwartz and J. Bruck, "Rank modulation for flash memories," in *Proc. IEEE International Symposium on Information Theory (ISIT)*, 2008, pp. 1731-1735.
[22] A. Jiang, R. Mateescu, E. Yaakobi, J. Bruck, P. Siegel, A. Vardy and J. Wolf, "Storage coding for wear leveling in flash memories," in *Proc. IEEE International Symposium on Information Theory (ISIT'09)*, Seoul, Korea, June-July 2009.
[23] A. Jiang, M. Schwartz and J. Bruck, "Error-correcting codes for rank modulation," in *Proc. IEEE International Symposium on Information Theory (ISIT)*, 2008, pp. 1736-1740.
[24] M. Kendall and J. D. Gibbons, *Rank correlation methods*. Oxford University Press, NY, 1990.
[25] A. V. Kuznetsov and B. S. Tsybakov, "Coding for memories with defective cells," *Problemy Peredachi Informatsii*, vol. 10, no. 2, pp. 52–60, 1974.
[26] D. H. Lehmer, "Teaching combinatorial tricks to a computer," in *Proc. Sympos. Appl. Math. Combinatorial Analysis*, vol. 10, Amer. Math. Soc., Providence, R.I., pp. 179-193, 1960.
[27] F. Merkx, "WOM codes constructed with projective geometries," *Traitment du Signal*, vol. 1, no. 2-2, pp. 227–231, 1984.
[28] 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.
[29] R. L. Rivest and A. Shamir, "How to reuse a 'write-once' memory," *Information and Control*, vol. 55, pp. 1-19, 1982.

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

[31] Z. Wang, A. Jiang and J. Bruck, "On the capacity of bounded rank modulation for flash memories," in *Proc. IEEE International Symposium on Information Theory (ISIT)*, Seoul, Korea, June-July 2009.

[32] F. M. J. Willems and A. J. Vinck, "Repeated recording for an optical disk," in *Proc. 7th Symp. Inform. Theory in the Benelux*, May 1986, Delft Univ. Press, pp. 49-53.

[33] 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.

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

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

Data Storage

Edited by Florin Balasa

ISBN 978-953-307-063-6

Hard cover, 226 pages

Publisher InTech

Published online 01, April, 2010

Published in print edition April, 2010

The book presents several advances in different research areas related to data storage, from the design of a hierarchical memory subsystem in embedded signal processing systems for data-intensive applications, through data representation in flash memories, data recording and retrieval in conventional optical data storage systems and the more recent holographic systems, to applications in medicine requiring massive image databases.

How to reference

In order to correctly reference this scholarly work, feel free to copy and paste the following:

Anxiao (Andrew) Jiang, Jehoshua Bruck (2010). Data Representation for Flash Memories, Data Storage, Florin Balasa (Ed.), ISBN: 978-953-307-063-6, InTech, Available from: http://www.intechopen.com/books/data-storage/data-representation-for-flash-memories

INTECH
open science | open minds

InTech Europe
University Campus STeP Ri
Slavka Krautzeka 83/A
51000 Rijeka, Croatia
Phone: +385 (51) 770 447
Fax: +385 (51) 686 166
www.intechopen.com

InTech China
Unit 405, Office Block, Hotel Equatorial Shanghai
No.65, Yan An Road (West), Shanghai, 200040, China
中国上海市延安西路65号上海国际贵都大饭店办公楼405单元
Phone: +86-21-62489820
Fax: +86-21-62489821

IntechOpen