Joint Coding for Flash Memory Storage

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

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

***Abstract*—Flash memory is an electronic non-volatile memory with wide applications. Due to the substantial impact of block erasure operations on the speed, reliability and longevity of flash memories, writing schemes that enable data to be modified numerous times without incurring the block erasure is desirable. This requirement is addressed by floating codes, a coding scheme that jointly stores and rewrites data and maximizes the rewriting capability of flash memories. In this paper, we present several new floating code constructions. They include both codes with specific parameters and general code constructions that are asymptotically optimal. We also present bounds to the performance of floating codes.**

I. Introduction

Flash memory is a type of electronic non-volatile memory (NVM) with wide applications. It stores data in floating-gate cells, where each cell has q states: $0, 1, \cdots, q-1$. To increase or decrease the state of a cell, charge is injected into or extracted from the cell using the hot-electronic mechanism or the Fowler-Nordheim tunneling mechanism [1]. An interesting feature of flash memories is their *block erasure* operation. Cells in a flash memory are organized into blocks, with each block containing 10^5 or so cells. The state of a cell can be raised individually (called *cell programming*). But to decrease the state of a cell, the flash memory needs to erase the whole block (i.e., lowering the states of all the cells to the minimum value) and then re-program all the cells. Such an operation is called *block erasure/programming*. It is well known that block erasures can substantially reduce the writing speed, reliability and longevity of flash memories [1], with the benefit of a lower circuitry complexity. For storage schemes, it is important to minimize block erasures, especially for applications where data are modified frequently.

Floating codes [4] address this requirement by maximizing the number of times data can be rewritten (i.e., modified) between two block erasures. A floating code jointly stores multiple variables in n cells and with each rewrite, the cells' states keep increasing. No block erasure is necessary until the cell states reach the maximum state value. A floating code maps the state of cells to the stored variables and, through the joint coding approach, the ability to support rewrites can be substantially improved. Floating codes generalize the well known WOM (write-once memory) codes (where a single variable is considered) [3], [6].

The known results on floating codes are limited. Existing code constructions are mainly for two to three binary variables [4]. In this paper, we present several new code constructions based on varied approaches. They include both codes with specific parameters and code constructions for general parameters. We analyze their performance and bounds, and show the asymptotic optimality of most of the presented codes.

II. Notation

Let $v_1, v_2, \cdots, v_k$ be k variables. Each variable v_i has an alphabet of size l: $\{0, 1, \cdots, l-1\}$. $(v_1, v_2, \cdots, v_k)$ is called the *variable vector*, and V denotes the set of all l^k variable vectors. Let $c_1, c_2, \cdots, c_n$ be the states of n cells. Each cell state c_i has one of q possible states: $\{0, 1, \cdots, q-1\}$. $(c_1, c_2, \cdots, c_n)$ is called the *cell state vector*, and C denotes the set of all q^n cell state vectors. The *weight* of a cell state vector is $\sum_{i=1}^{n} c_i$.

Definition 1. *[4] A floating code is a mapping $D : C \to V \cup \{\perp\}$. For $\alpha \in C$ and $\beta \in V$, if $D(\alpha) = \beta$, it means that the cell states α represent the variable values β; if $D(\alpha) = \perp$, it means that α does not represent any value of the variables.* □

A *rewrite* means to change the value of one of the k variables. Initially, the cell state vector is $(0, 0, \cdots, 0)$, and they represent the variable vector $(0, 0, \cdots, 0)$. A floating code is for rewriting data between two block erasures; so for each rewrite, the state of a cell can only increase or remain the same. A rewrite changes the cell states so that the new cell state vector represents the new variable vector. Let t denote the number of rewrites that are guaranteed to be feasible by using the floating code, regardless of what the sequence of rewrites are. Clearly, t is a finite number. Given the parameters n, q, k, l, a floating code that maximizes t is called *optimal*.

Example 2. *A floating code for $n = k = 5, q = 4, l = 2$ is shown in Fig. 1. The code has a cyclic property: If the cell state vector $(c_1 = a_1, c_2 = a_2, \cdots, c_n = a_n)$ represents the variable vector $(v_1 = b_1, v_2 = b_2, \cdots, v_k = b_k)$, then the cell state vector $(c_1 = a_2, c_2 = a_3, \cdots, c_{n-1} = a_n, c_n = a_1)$ represents the variable vector $(v_1 = b_2, v_2 = b_3, \cdots, v_{k-1} = b_k, v_k = b_1)$. For simplicity, for every set of cell state vectors that are cyclic shifts of each other, only one of them is shown in Fig. 1 as their representative.*

If a sequence of rewrites change the variables as $(0,0,0,0,0) \to (1,0,0,0,0) \to (1,0,1,0,0) \to (1,0,0,0,0) \to (1,0,0,0,1) \to (1,0,1,0,1) \to (1,0,1,1,1)$, the cell state vector can change as $(0,0,0,0,0) \to (1,0,0,0,0) \to (1,0,1,0,0) \to (2,1,1,1,1) \to (2,1,1,1,2) \to (2,1,2,1,2) \to (2,1,2,2,2)$. A general code construction for

978-1-4244-2571-6/08/$25.00 ©2008 IEEE

$n = k, l = 2, t = 2(q-1)$, *including this code as a special example, is shown in Section III.* □

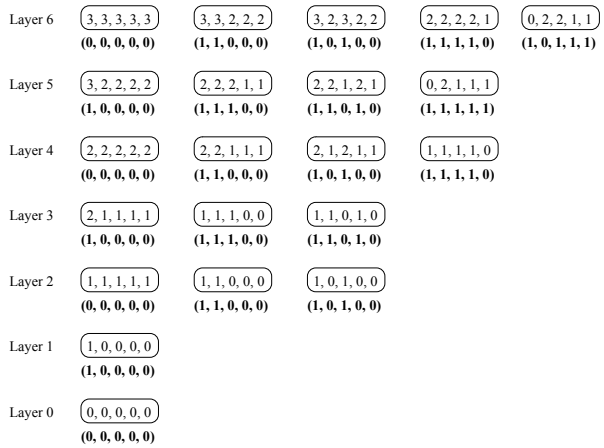

Figure 1. A floating code with $n = k = 5, l = 2, q = 4, t = 6$. The numbers inside (resp. beside) a node are the cell state vector (resp., variable vector). The code has a *cyclic property*. The *layer* of a cell state vector is the number of rewrites it takes for the cells to each that state.

We present an upper bound to t for general floating codes.

Theorem 3. *For $i = 1, 2, \cdots, k$, define s_i as follows: (1) If $l = 2$ and i is even, $s_i = \sum_{j=0,2,\cdots,i} \binom{k}{j}$; (2) if $l = 2$ and i is odd, $s_i = \sum_{j=1,3,\cdots,i} \binom{k}{j}$; (3) if $l > 2$, $s_i = \sum_{j=0}^{i} \binom{k}{j}(l-1)^j$. Also, define w_i as the smallest positive integer such that $\binom{n+w_i}{n} - \binom{n+i-1}{n} \geqslant s_i$. Let $m \in \{1, 2, \cdots, k\}$ be the integer such that $\frac{w_m}{m} \geqslant \frac{w_j}{j}$ for $j = 1, 2, \cdots, k$, .*

For any floating code, $t \leqslant \lfloor \frac{n(q-1)}{w_m} \rfloor \cdot m + \min\{m-1, n(q-1) \bmod w_m\}$.

Proof: s_i is the number of values that the variable vector can possibly take after i rewrites. (By symmetry, s_i does not depend on the initial value of the variable vector.) Consider m consecutive rewrites. They increase the weight of the cell state vector by at least m. For any $x \geqslant m$, the number of ways to raise the states of n cells such that the weight of the cell state vector increases by at least m and at most x is $\binom{n+x}{n} - \binom{n+m-1}{n}$. The m consecutive rewrites can change the variable into s_m possible values, each of which corresponds to at least one way of raising the cell states. So by the definition of w_i, there is a sequence of m consecutive rewrites that increases the weight of the cell state vector by at least w_m. Choose $\lfloor \frac{n(q-1)}{w_m} \rfloor$ such sequences of rewrites (one after another), and they make the weight of the cell state vector be at least $n(q-1) - [n(q-1) \bmod w_m]$. After that, since the weight cannot exceed $n(q-1)$, there is a sequence of m rewrites that is not feasible due to the lack of room for the weight increase. Also, each rewrite increases the weight by at least one. So after the initial $\lfloor \frac{n(q-1)}{w_m} \rfloor$ sequences of rewrites (which consist of $\lfloor \frac{n(q-1)}{w_m} \rfloor \cdot m$ rewrites in total), at most $\min\{m-1, n(q-1) \bmod w_m\}$ more rewrites are guaranteed to be feasible. So $t \leqslant \lfloor \frac{n(q-1)}{w_m} \rfloor \cdot m + \min\{m-1, n(q-1) \bmod w_m\}$. ■

The bound in Theorem 3 compares favorably with the upper bounds in [4] when k or l is relatively large. For example, when $n = 4, q = 8, k = 4, l = 4$, Theorem 3 gives $t \leqslant 11$, and the bounds in [4] show $t \leqslant 14$.

III. Code Construction for $n = k$

In this section, we present a floating code for $n = k \geqslant 3$, $l = 2$ and arbitrary q. Codes for more general parameters will be presented in the following sections. It will be proved that when $n = k = 3$, the code presented here is optimal. An example of the code has been shown in Example 2 and Fig. 1. We now present the general code construction. (Note that the code has a *cyclic property*, as explained in Example 2.) In the following, define $s_{min} = \min\{c_1, c_2, \cdots, c_n\}$ and $s_{max} = \max\{c_1, c_2, \cdots, c_n\}$.

Construction 4. *(Code for $n = k \geqslant 3$, $l = 2$ and arbitrary q) The cell state vectors $(c_1, c_2, \cdots, c_n)$ are mapped to the variable vectors $(v_1, v_2, \cdots, v_k)$ in the following way:*

- Type I: *If $c_1 = c_2 = \cdots = c_n$, then $v_i = 0$ for $1 \leqslant i \leqslant k$.*
- Type II: *If $s_{max} = s_{min} + 1$, then $v_i = c_i - s_{min}$ for $1 \leqslant i \leqslant k$.*
- Type III: *If $(c_1, c_2, \cdots, c_n) = (s_{min}, s_{min} + 2, s_{min} + 1, s_{min} + 1, \cdots, s_{min} + 1)$ – that is, it starts with $s_{min}, s_{min} + 2$ and is followed by $n - 2$ $s_{min} + 1$'s – then $v_i = 1$ for $1 \leqslant i \leqslant k$.*
- Type IV: *If $(c_1, c_2, \cdots, c_n) = (s_{min}, s_{min} + 2, s_{min} + 2, s_{min} + 1, s_{min} + 1, \cdots, s_{min} + 1)$ – that is, it starts with $s_{min}, s_{min} + 2, s_{min} + 2$ and is following by $n - 3$ $s_{min} + 1$'s – then $v_2 = 0$ and $v_i = 1$ for $i \neq 2, 1 \leqslant i \leqslant k$.*
- Cyclic law: *If we cyclically shift any cell state vector mentioned above by i positions ($0 \leqslant i \leqslant n-1$), the corresponding variable vector also cyclically shifts by i positions.* □

To explain how the code is used for rewriting, let's first define *layer number*. Every cell state vector has a layer number (see Fig. 1 for examples), which is the number of rewrites it takes for the cells to reach that state. For notational convenience, if C is a cell state vector of type I (resp., II, III or IV), then we call a cyclic shift of C type I (resp., II, III or IV) as well. For this code, the layer number of a cell state vector $\alpha = (c_1, c_2, \cdots, c_n)$ is determined as follows:

- If α is of type I, it is in layer $2c_1$.
- If α is of type II, let x denote the number of cells whose states are s_{max}, then it is in layer $2s_{min} + x$.
- If α is of type III, it is in layer $2s_{min} + n$.
- if α is of type IV, it is in layer $2s_{min} + n + 1$.

For a cell state vector in layer $i \geqslant 0$, a rewrite always changes it into a cell state vector in layer $i + 1$. For example, if $n = k = 5$ and the current cell state vector is $(0, 2, 1, 1, 1)$ (type III, layer 5, representing $(v_1, v_2, \cdots, v_5) = (1, 1, 1, 1, 1)$), and we want to change the variable vector to $(1, 1, 1, 0, 1)$, we can change the variable state vector to $(2, 2, 2, 1, 2)$ (type II, layer 6). It is simple to verify through case enumeration that for any cell state vector in layer $i < 2(q-1)$, there are k cell state vectors in layer $i + 1$

it can change into that correspond to the k possible rewrite requests. So we get:

Theorem 5. *The code in Construction 4 has $t = 2(q-1)$.*

Corollary 6. *When $k = 3$, the code is optimal.*

Proof: By the Theorem 2 of [4], for any floating code with $n \geqslant k(l-1)-1$, $t \leqslant [n-k(l-1)+1](q-1) + \lfloor \frac{[k(l-1)-1](q-1)}{2} \rfloor$. So when $n = k = 3$ and $l = 2$, $t \leqslant 2(q-1)$. That matches the performance of this code. ■

IV. Composite Codes with $3 \leqslant k \leqslant 6$

In this section, we present a family of codes for $3 \leqslant k \leqslant 6$ and $l = 2$. Due to the space limitation, we present the detailed code construction for $k = 4$ and briefly summarize the codes for $k = 3, 5, 6$. (Interested readers please refer to [5].)

Construction 7. *(Code for $k = 4$, $l = 2$, $n \geqslant 7$ and arbitrary q) We first show the construction for a simplified case: $q = 2$. Here every valid cell state vector (i.e., a cell state vector that represents variables) has at least three cells at state 0. The segment of cells before (resp., behind) the second (resp., second-last) cell at state 0 is called the head (resp. tail). (For example, if the cell state vector is $(0,1,0,0,1,1)$, then the head is $(0,1)$, the tail is $(0,1,1)$.) For simplicity of explanation, denote the cell state vector by $(a_1, a_2, \cdots, a_i, 0, \cdots, 0, b_j, \cdots, b_2, b_1)$. Here the head and the tail have length i and j, respectively. Note that both of them contain exactly one cell at state 0.*

The two variables v_1, v_2 are determined by the head as follows: (1) $v_1 = v_2 = 0$ if i is odd and $a_i = 0$; (2) $v_1 = v_2 = 1$ if i is odd and $a_i \neq 0$; (3) $v_1 = 0, v_2 = 1$ if i is even and $a_i \neq 0$; (4) $v_1 = 1, v_2 = 0$ if i is even and $a_i = 0$.

The two variables v_3, v_4 are determined by the tail in the same way. Note that here b_1 replaces a_1, b_2 replaces a_2, and so on; and v_3 (resp. v_4) replaces v_1 (resp., v_2).

For a rewrite, if we need to modify the head (resp., tail), we always change the leftmost (resp., rightmost) cell that gives the desired result. A rewrite changes the state of exactly one cell. The process ends when only three cells at state 0 are left.

If $q > 2$, we use the cell states layer-by-layer: first use cell states 0 and 1 as above; then use cell levels 1 and 2 in the same way; then use cell levels 2 and 3 $\cdots$ Each rewrite raises only one cell's state except during the transition from one layer to the next. □

Example 8. *Let $k = 4, l = 2, n = 7, q = 4$. If the variable vector changes as $(0,0,0,0) \to (1,0,0,0) \to (1,1,0,0) \to (1,1,1,0) \to (0,1,1,0) \to (0,1,0,0) \to (0,1,0,1) \to \cdots$, the cell states change as $(0,0,0,0,0,0,0) \to (1,0,0,0,0,0,0) \to (1,0,1,0,0,0,0) \to (1,0,1,0,0,0,1) \to (1,0,1,1,0,0,1) \to (1,2,1,1,1,1,1) \to (1,2,1,1,1,2,1) \to \cdots$* □

Theorem 9. *For the code in Construction 7, if n is even, $t = (n-6)(q-1)+3$; if n is odd, $t = (n-5)(q-1)+2$.*

Proof: It is not hard to see that every rewrite raises one cell's state by one, unless the rewrite causes the transition from one layer to the next. During that transition, if n is even (resp., odd), at most four (resp., three) cells need to be set to the higher state of the new layer. So the first layer supports $n-3$ rewrites and every subsequent layer supports at least $n-6$ (if n is even) or $n-5$ (if n is odd) rewrites. ■

The following results summarize the codes for $k = 3, 5$ and 6. (For details of the code constructions, please see [5].)

- When $k = 3, l = 2$, $n \geqslant 5$, if n is even, there is a code with $t = (n-4)(q-1)+2$; otherwise, there is a code with $t = (n-3)(q-1)+1$.
- When $k = 5, l = 2$, $n \geqslant 9$, there is a code with $t > (n-10-2\log_2 n)(q-1)+3$.
- When $k = 6, l = 2$, $n \geqslant 12$, there is a code with $t > (n-17-6\log_2 n)(q-1)+5$.

All the four codes presented here have $t = n(q-1) - o(nq)$. Since every rewrite raises the cell states (up to $q-1$), the codes are all asymptotically optimal in n, the number of cells, and in q, the number of cell levels.

V. Indexed Code

The codes introduced above are for the joint coding of a few variables. In this section, we introduce a code construction, *indexed code*, for general k.

Construction 10. *(Indexed code) Divide the k variables into a groups: $g_1, g_2, \cdots, g_a$. For the n cells, set aside a small number of cells as index cells and divide the other cells into b groups: $h_1, h_2, \cdots, h_b$. Here a and b are chosen parameters, and $b \geqslant a$. For $1 \leqslant i \leqslant a$, the variables of g_i are coded using a floating code and are stored in h_i. Afterwards, every time a cell group can no longer support any more rewriting (say it stores g_i), store g_i in the next unused cell group. The index cells are used to remember which cell group stores which variable group. (The details of the index cells is the topic of study in this section.)* □

We first show that the indexed code is asymptotically optimal in n and q. Let there be $\sqrt{n} - o(\sqrt{n})$ cell groups, each containing $\sqrt{n} - o(\sqrt{n})$ cells. At most $log_q a$ index cells are needed on average per cell group. Apply known floating codes, such as those presented in Section IV, to each variable group. Those codes can support $\sqrt{n}(q-1) - o(\sqrt{n}q)$ rewrites in each cell group. There can be at most a partially used cell groups at any moment, so in the end, $\sqrt{n} - o(\sqrt{n})$ cell groups are fully used. So the indexed code enables $n(q-1) - o(nq)$ rewrites.

The simplest way to use index cells is to use $\log_q a$ index cells for each cell group to remember which variable group it stores. However, when n is sufficiently large, much fewer index cells are necessary. The best coding strategy is to remember the mapping between the a partially used cell groups and the a variable groups, which is a permutation. (The unused, partially used, and fully used cell group are differentiated in the following way. Cells in unused groups are at state 0. Make cells in a fully used group all have state $q-1$. Ensure that in a partially used group, at least one cell is not at state $q-1$. The last step costs the support for at most one rewrite, depending on the used floating code.)

Example 11. *Let $a = 3$ and $b = 6$. Initially, the cell group h_i stores the variable group g_i for $i = 1,2,3$. Assume that as rewriting continues, first h_4 is used to store g_2, then h_5 is used to store g_1, then h_6 is used to store g_2. We use a permutation (π_1, π_2, π_3) to record the information that the i-th partially used cell group stores the π_i-th variable group, for $i = 1,2,3$. Then, the permutation changes as $(1,2,3) \to (1,3,2) \to (3,2,1) \to (3,1,2)$. The index cells are used to remember the permutation. Note that the permutation changes at most $b-a$ times.*

We now show a coding strategy for the index cells. First, build a mapping between the permutations and binary vectors of length four as follows.

binary vector	(0,0,0,0) (0,1,1,1)	(0,0,0,1) (0,1,1,0)	(0,0,1,0) (0,1,0,1)	(1,0,1,0) (1,1,0,1)	(1,0,0,0) (1,1,1,1)	(1,0,0,1) (1,1,1,0)
permutation	(1,2,3)	(3,1,2)	(2,3,1)	(2,1,3)	(1,3,2)	(3,2,1)

For every change of the permutation, only one bit in the binary vector needs to change. (Note that every such change shifts one number to the end of the permutation.) For instance, if the permutation changes as $(1,2,3) \to (1,3,2) \to (3,2,1) \to (2,1,3) \to (2,3,1)$, the binary vector can change as $(0,0,0,0) \to (1,0,0,0) \to (1,0,0,1) \to (1,1,0,1) \to (0,1,0,1)$. Use a floating code that stores the four bits in the binary vector (such as the code for four variables in Section IV), and this code also records the permutation. By Theorem 9, only $\frac{b}{q-1} + o(b)$ index cells are needed, which is close to one index cell on average for every $q-1$ cell groups when b is large. □

The design of general coding schemes for index cells is beyond the scope of this paper. In the following, We present a lower bound for the number of index cells that are needed for remembering the permutation (i.e., the mapping between partially used cell groups and the variable groups).

Theorem 12. *The number of cells needed for indexing is at least $\frac{b-a}{q-1} + \frac{a}{2} - 1$ if $(a-2)(q-1) < 2(b-a)$, and at least $\frac{2(b-a)}{q-1}$ if $(a-2)(q-1) \geqslant 2(b-a)$.*

Proof: Assume that an indexing scheme that uses x cells for indexing is given. We first prove that they can only guarantee the recording of at most $(x-a+2)(q-1) + \frac{(a-2)(q-1)}{2}$ changes of the permutation if $x > a-2$, and at most $\frac{x(q-1)}{2}$ changes of the permutation if $x \leqslant a-2$.

Let's consider the case $x > a-2$ first. Let $(s_1, s_2, \cdots, s_x)$ denote the states of the x cells. Initially, $(s_1, s_2, \cdots, s_x) = (0,0,\cdots,0)$ and the permutation is $(1,2,\cdots,a)$. Let $A = (s_1, s_2, \cdots, s_{a-2})$ denote the first $a-2$ cells' states, and let $B = (s_{a-1}, s_a, \cdots, s_x)$ denote the last $x-a+2$ cells' states. Define the *weight of* A (resp., B) as $\sum_{i=1}^{a-2} s_i$ (resp., $\sum_{i=a-1}^{x} s_i$).

The permutation is a permutation of a numbers; and when it changes, a number is moved to the back. So there are $a-1$ possible ways to change a permutation each time. When the permutation changes, some cell states need to be raised. Since there are only $a-2$ cells in A, at any time, if all the $a-1$ ways to change the permutation increase only the weight of A (not the weight of B), there must be one way of changing the permutation that increases the weight of A by at least two.

We now choose a sequence of changes to the permutation as follows. Assume that $i \geqslant 0$ changes have been chosen. For the $(i+1)$-th change, if all the $a-1$ ways to change the permutation increase only the weight of A (not the weight of B), choose the $(i+1)$-th change as one that increases the weight of A by at least two (we call such a change *type I*); otherwise, choose the $(i+1)$-th change as one that increases the weight of B by at least one (we call such a change *type II*). Since the maximum weight of A is $(a-2)(q-1)$ and the maximum weight of B is $(n-a+2)(q-1)$, there can be at most $\frac{(a-2)(q-1)}{2}$ changes of type I and at most $(n-a+2)(q-1)$ changes of type II. So the number of changes of permutation that the indexing scheme guarantees to record is at most $(x-a+2)(q-1) + \frac{(a-2)(q-1)}{2}$.

The case $x \leqslant a-2$ is simpler. By the same argument, we can choose a sequence of changes of the permutation such that every change increases $\sum_{i=1}^{x} s_i$ by at least two. So the number of changes of permutation that the indexing scheme guarantees to record is at most $\frac{x(q-1)}{2}$.

Since there are a variable groups and b cell groups, there can be $b-a$ changes of permutation. So an indexing scheme needs to have $(x-a+2)(q-1) + \frac{(a-2)(q-1)}{2} \geqslant b-a$ if $x > a-2$ and have $\frac{x(q-1)}{2} \geqslant b-a$ if $x \leqslant a-2$. Thus we get $x \geqslant \frac{b-a}{q-1} + \frac{a}{2} - 1$ if $x > a-2$ and $x \geqslant \frac{2(b-a)}{q-1}$ if $x \leqslant a-2$. So when $\frac{(a-2)(q-1)}{2} < b-a$, we must have $x > a-2$ and therefore have $x \geqslant \frac{b-a}{q-1} + \frac{a}{2} - 1$. When $\frac{(a-2)(q-1)}{2} \geqslant b-a$, we either have $x > a-2 \geqslant \frac{2(b-a)}{q-1}$, or have $x \leqslant a-2$ and therefore have $x \geqslant \frac{2(b-a)}{q-1}$. So the conclusion holds. ■

VI. Constructions Based on Covering Codes

There have been no existing floating code constructions for $l > 2$ (i.e., non-binary alphabets) [4]. In this section, we present a new method that converts floating codes with large alphabets to floating codes with small alphabets (including the binary alphabet) by using covering codes. The idea is to map a variable with a large alphabet to a vector of a small alphabet such that when the variable changes its value (i.e., is rewritten), only a few (preferably one) entries in the vector change their values. Based on this method, we can obtain a series of bounds and code constructions for large alphabets.

Construction 13. *(Mapping based on linear covering codes) Let v be a variable of alphabet size l. Choose an (n_0, k_0) linear covering code of alphabet size l_0, which has length n_0 and dimension k_0. The requirement is $l_0^{n_0-k_0} \geqslant l$. The code has $l_0^{n_0-k_0}$ cosets of the codewords. Among them, choose any l cosets, and map them to the l values of v.* □

Example 14. *Let v be a variable that takes its value from an alphabet of size $l = 4$: $\{0,1,2,3\}$. Choose the simple $(3,1)$ repetition code. As a result, the mapping from v to bit vectors of length 3 is as follows:*

vector	*(0,0,0)* *(1,1,1)*	*(1,0,0)* *(0,1,1)*	*(0,1,0)* *(1,0,1)*	*(0,0,1)* *(1,1,0)*
v	*0*	*1*	*2*	*3*

To design a floating code for variables $v_1, v_2, \cdots, v_k$ of alphabet size 4, we first map them to binary variables $\{w_{i,j} | 1 \leqslant i \leqslant k, 1 \leqslant j \leqslant 3\}$, where each binary vector $(w_{i,1}, w_{i,2}, w_{i,3})$ represents v_i. Then we use a floating code for the $3k$ binary variables. Every rewrite for $(v_1, v_2, \cdots, v_k)$ maps to exactly one rewrite for $(w_{1,1}, w_{1,2}, \cdots, w_{k,3})$. (For instance, if $k = 2$ and (v_1, v_2) changes as $(0,0) \to (0,3) \to (0,2) \to (3,2) \to (3,1)$, the binary vector $(w_{1,1}, w_{1,2}, w_{1,3}, w_{2,1}, w_{2,2}, w_{2,3})$ will correspondingly change as $(0,0,0,0,0,0) \to (0,0,0,0,0,1) \to (0,0,0,1,0,1) \to (0,0,1,1,0,1) \to (0,0,1,1,0,0)$.) So if the floating code supports t rewrites for the binary variables, it also supports t rewrites for the 4-ary variables $v_1, v_2, \cdots, v_k$. □

It is important for the selected linear covering code to have a small covering radius, because when the large-alphabet variable changes, the covering radius of the code equals the number of entries in the small-alphabet vector that may change.

Let R denote the covering radius of the (n_0, k_0) covering code in Construction 13. Let $t(n,q,k,l)$ denote the greatest number of rewrites that a floating code can guarantee to support, when k l-ary variables are stored in n cells with q states. (Namely, $t(n,q,k,l)$ is the optimal value of t for floating codes with parameters n,q,k,l.) The following theorem compares the coding performance for different alphabets.

Theorem 15.

$$t(n,q,k,l) \geqslant \lfloor t(n,q,kn_0,l_0)/R \rfloor$$

Proof: Map the variables $v_1, v_2, \cdots, v_k$ of alphabet size l to kn_0 variables of alphabet size l_0 with Construction 13. Build an optimal floating code C for the kn_0 variables of alphabet size l_0, which guarantees $t(n,q,kn_0,l_0)$ rewrites.

For the (n_0, k_0) covering code, every vector of length n_0 is within Hamming distance R from a codeword. So by the symmetry of linear codes, for every vector and each of the $l_0^{n_0-k_0}$ cosets, there is a vector in the coset that is within Hamming distance R from the former vector. So when we rewrite v_i $(1 \leqslant i \leqslant k)$, we are correspondingly rewriting at most R l_0-ary variables. So C supports $\lfloor t(n,q,kn_0,l_0)/R \rfloor$ rewrites for $v_1, v_2, \cdots, v_k$. So $t(n,q,k,l) \geqslant \lfloor t(n,q,kn_0,l_0)/R \rfloor$. ■

By using known covering codes [2], we can obtain a number of bounds for floating codes with large alphabets in terms of the performance of floating codes with binary alphabets. We report some of the results in Fig. 2. (For the full set of results, please refer to [5].) Since there has been a number of floating code constructions for binary variables (especially the codes presented in this paper), floating codes with large alphabets can also be built. Due to the space limit, we report the codes in [5].

1. For $m \geqslant 2$, $l \leqslant 2^m$, $t(n,q,k,l) \geqslant t(n,q,k(2^m-1),2)$.
2. For $l \leqslant 2^{11}$, $t(n,q,k,l) \geqslant \lfloor t(n,q,23k,2)/3 \rfloor$.
3. For $a \geqslant b \geqslant 1$ and $l \leqslant 2^{a-b}$, $t(n,q,k,l) \geqslant \lfloor t(n,q,ka,2)/\lceil \frac{a-b}{2} \rceil \rfloor$.
4. For $b \geqslant 4$, $a \geqslant 2^{b-2}$ and $l \leqslant 2^{a-b}$, $t(n,q,k,l) \geqslant \lfloor t(n,q,ka,2)/(\lfloor \frac{a}{2} \rfloor - 2^{(b-4)/2}) \rfloor$.
5. For $l \leqslant 2^7$, $t(n,q,k,l) \geqslant \lfloor t(n,q,23k,2)/2 \rfloor$.
6. For $l \leqslant 2^{19}$, $t(n,q,k,l) \geqslant \lfloor t(n,q,47k,2)/5 \rfloor$.
7. For all $a \geqslant 1$ and $l \leqslant 2^{a-1}$, $t(n,q,k,l) \geqslant \lfloor t(n,q,ka,2)/\lfloor \frac{a}{2} \rfloor \rfloor$.
8. For all $a \geqslant 2$ and $l \leqslant 2^{a-2}$, $t(n,q,k,l) \geqslant \lfloor t(n,q,ka,2)/\lfloor \frac{a-1}{2} \rfloor \rfloor$.
9. For all $a \geqslant 3$ and $l \leqslant 2^{a-3}$, $t(n,q,k,l) \geqslant \lfloor t(n,q,ka,2)/\lfloor \frac{a-2}{2} \rfloor \rfloor$.
10. For $a \geqslant 6$, $l \leqslant 2^{a-4}$, $t(n,q,k,l) \geqslant \lfloor t(n,q,ka,2)/\lfloor \frac{a-4}{2} \rfloor \rfloor$.
11. For $a \geqslant 7$, $l \leqslant 2^{a-5}$, $t(n,q,k,l) \geqslant \lfloor t(n,q,ka,2)/\lfloor \frac{a-5}{2} \rfloor \rfloor$.
12. For $a \geqslant 14$, $l \leqslant 2^{a-6}$, $t(n,q,k,l) \geqslant \lfloor t(n,q,ka,2)/\lfloor \frac{a-8}{2} \rfloor \rfloor$
13. For $a \geqslant 19$, $l \leqslant 2^{a-7}$, $t(n,q,k,l) \geqslant \lfloor t(n,q,ka,2)/\lfloor \frac{a-9}{2} \rfloor \rfloor$.
14. For all $b \geqslant 2$, $a \geqslant 2^{2b}-1$ and $l \leqslant 2^{a-2b-1}$, $t(n,q,k,l) \geqslant \lfloor t(n,q,ka,2)/\lfloor \frac{a-2^b}{2} \rfloor \rfloor$.
15. For all $b \geqslant 2$, a even and $l \leqslant 2^{a-2b}$, $t(n,q,k,l) \geqslant \lfloor t(n,q,ka,2)/\lfloor \frac{a-2^{(2b-1)/2}}{2} \rfloor \rfloor$.
16. For all $b \geqslant 2$, a odd and $l \leqslant 2^{a-2b}$, $t(n,q,k,l) \geqslant \lfloor t(n,q,ka,2)/\lfloor \frac{a-2^{(2b-1)/2}-1}{2} \rfloor \rfloor$.
17. For $a \geqslant 127$ and $l \leqslant 2^{a-8}$, $t(n,q,k,l) \geqslant \lfloor t(n,q,ka,2)/\lfloor \frac{a-16}{2} \rfloor \rfloor$.
18. For all $m \geqslant 3$ and $l \leqslant 2^{2m+1}$, $t(n,q,k,l) \geqslant \lfloor t(n,q,k(2^{m+1}+2^m-4),2)/2 \rfloor$.
19. For all $m \geqslant 4$ and $l \leqslant 2^{2m}$, $t(n,q,k,l) \geqslant \lfloor t(n,q,k(2^{m+1}-4),2)/2 \rfloor$.
20. For all $m \geqslant 1$ and $l \leqslant 2^{4m}$, $t(n,q,k,l) \geqslant \lfloor t(n,q,k(2^{2m+1}-2^m-1),2)/2 \rfloor$.
21. For all $m \geqslant 2$ and $l \leqslant 2^{4m+1}$, $t(n,q,k,l) \geqslant \lfloor t(n,q,k(2^{2m+1}+2^{2m}-2^m-2),2)/2 \rfloor$.
22. For all $m \geqslant 2$ and $l \leqslant 2^{4m+2}$, $t(n,q,k,l) \geqslant \lfloor t(n,q,k(2^{2^{2m+2}-2^m-2}),2)/2 \rfloor$.
23. For all $m \geqslant 2$ and $l \leqslant 2^{4m+3}$, $t(n,q,k,l) \geqslant \lfloor t(n,q,k(2^{2m+2}+2^{2m+1}-2^m-2),2)/2 \rfloor$.

Figure 2. The relationship between floating codes with $l > 2$ and floating codes with $l = 2$. Here $t(n,q,k,l)$ denotes the optimal value of t (the number of rewrites) for a floating code with the parameters n,q,k,l.

REFERENCES

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

[2] G. Cohen, I. Honkala, S. Litsyn and A. Lobstein. *Covering codes*, North-Holland, 1997.

[3] 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. Information Theory*, vol. 45, no. 1, pp. 308-313, 1999.

[4] A. Jiang, V. Bohossian and J. Bruck, "Floating codes for joint information storage in write asymmetric memories," *Proc. IEEE International Symposium on Information Theory (ISIT)*, Nice, France, June 2007.

[5] A. Jiang and J. Bruck, "Joint coding for flash memory storage," manuscript, *http://faculty.cs.tamu.edu/ajiang/jc.pdf*.

[6] R. L. Rivest and A. Shamir, "How to reuse a 'write-once' memory," *Information and Control*, vol. 55, pp. 1-19, 1982.