Floating Codes for Joint Information Storage in Write Asymmetric Memories

Anxiao (Andrew) Jiang
Computer Science Department
Texas A&M University
College Station, TX 77843-3112
ajiang@cs.tamu.edu

Vasken Bohossian
Electrical Engineering Department
California Institute of Technology
Pasadena, CA 91125
vincent@paradise.caltech.edu

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

***Abstract*— Memories whose storage cells transit irreversibly between states have been common since the start of the data storage technology. In recent years, flash memories and other non-volatile EEPROM's based on floating-gate cells have become a very important family of such memories. We model them by the Write Asymmetric Memory (WAM), a memory where each cell is in one of q states – state 0, 1, $\cdots$, $q-1$ – and can only transit from a lower state to a higher state. Data stored in a WAM can be rewritten by shifting the cells to higher states. Since the state transition is irreversible, the number of times of rewriting is limited. When multiple variables are stored in a WAM, we study codes, which we call *floating codes*, that maximize the total number of times the variables can be written and rewritten.**

In this paper, we present several families of floating codes that are either optimal or asymptotically optimal. We also present bounds to the performance of general floating codes. The results show that floating codes can integrate a WAM's rewriting capabilities for different variables to a surprisingly high degree.

I. Introduction

Memories whose storage cells transit irreversibly between states have been common since the beginning of the data storage technology. Examples include punch cards and digital optical discs, where a cell can change from a 0-state to a 1-state but not *vice versa*. In recent years, flash memories and some other non-volatile EEPROM's based on floating-gate cells have become a very important family of such memories. They have good properties including high data density, fast reading time, physical robustness, etc., and have been widely used in mobile, mass as well as standard storage devices.

We use flash memories as a typical example to explain the basic storage mechanisms based on floating-gate cells. A flash memory consists of floating-gate cells as its basic storage elements. In most products, a cell has two states; but to increase data density, multi-level storage (where a cell has 4 to 256 or even more states) is being developed. For a cell with q states, we denote its states by $0, 1, \cdots, q-1$. To write (program) a cell, the hot-electron injection mechanism or the Fowler-Nordheim tunneling mechanism is used to inject electrons into the cell, where the electrons become trapped. The number of trapped electrons in a cell determines the threshold voltage of the cell: the more electrons, the higher the threshold voltage. The number of trapped electrons is chosen to concentrate around q discrete levels, corresponding to the q cell states. The state of a cell can be read by measuring the threshold voltage. Programming and reading cells are fast; however, rewriting data is much more complex. Most of the time, it requires moving cells to lower states for rewriting data, which means to remove electrons from the cells. In flash memories, cells are organized into blocks. A typical block using binary cells stores 64, 128 or 256 kilobytes of data. Due to circuit complexity reasons, to rewrite, first the whole block has to be erased (which means to lower all the cells of the block to the 0-state), then all the cells are reprogrammed. This happens even if only one cell really needs to lower its state for the rewriting, and it leads to a writing speed about 10^5 times slower than reading. Therefore, it will be very beneficial to design codes for storing data such that the data can be rewritten many times before the block has to be erased. Reducing the number of block erasing operations is critical not only for improving rewriting speed, but also for the flash memory's lifetime. Every erasing reduces the quality of the cells, and currently, a flash memory's lifetime is bounded by about 10^5 program-erase cycles. Although technically speaking, a cell can return to a lower state through block erasing, in this paper, we are interested in the writing and rewriting of data between two block erasing operations. In that period, the cells can only go from lower states to higher states.

We model the memories mentioned above using the following Write Asymmetric Memory (WAM) model. A WAM consists of n cells, where each cell has q states: state $0, 1, \cdots, q-1$. Such a cell is called a q-ary cell. A cell can go from state i to state j if and only if $i < j$.

WAM is a straightforward generalization of the Write Once Memory (WOM) model, firstly proposed by Rivest and Shamir [9], where $q = 2$. WAM is also a special case of the Generalized WOM model [3], where the state transition diagram of a cell can be any directed acyclic graph.

Research has been done on (generalized) WOM codes, where a single variable is stored in a WOM, and the code enables the variable to be rewritten numerous times. In practice, a memory stores many – let's say k – words. A simple approach to use the WOM codes in a memory is to partition it into k parts, where each part stores a word independently.

This simple approach, however, has a serious limitation. If the sequence of rewriting is very nonuniform across the words, which is common in many applications, the WAM becomes

unusable very soon. For example, say that each storage part allows t times of rewriting of a word. Once one of the k words needs rewriting for the $(t+1)$-th time, the WAM can no longer meet the requirement, even if the other $k-1$ words have not been rewritten yet. Therefore, it will be very beneficial to integrate the rewriting capabilities of the words, so that the words can be rewritten many times regardless of what the rewriting sequence is. As we will show in this paper, such an integration is feasible, many times to a surprisingly high degree. We call the codes that achieve it the *Floating Codes*.

We formally define the problem we study as follows. k variables are stored in a WAM, where each variable takes its value from an alphabet of size l: $\{0, 1, \cdots, l-1\}$. The WAM has n q-ary cells. Initially, all the cells are in the 0-state, and all the variables have the default value 0. Each rewriting updates the value of one variable. We use $(v_1, v_2, \cdots, v_k)$ – which we call the *variable vector* – to denote the values of the k variables, where $v_i \in \{0, 1, \cdots, l-1\}$. We use $(c_1, c_2, \cdots, c_n)$ – which we call the *cell state vector* – to denote the states of the n cells, where $c_i \in \{0, 1, \cdots, q-1\}$. A cell state vector $(c_1, c_2, \cdots, c_n)$ is said to be *above* another cell state vector $(c'_1, c'_2, \cdots, c'_n)$ if $c_i \geq c'_i$ for all i. When the cells change their states, they can only change to a state vector above the current one.

A *floating code* has two functions, $\alpha : \{0, 1, \cdots, q-1\}^n \rightarrow \{0, 1, \cdots, l-1\}^k$, and $\beta : \{0, 1, \cdots, q-1\}^n \times \{1, 2, \cdots, k\} \times \{0, 1, \cdots, l-1\} \rightarrow \{0, 1, \cdots, q-1\}^n$. Function α maps each cell state vector to a variable vector, which is used to decode (interpret) the stored data. Function β shows how to rewrite: given the current cell state vector and the information on which of the k variables is to be updated to which new value, the function β outputs the new cell state vector. The new cell state vector should correspond to the new values of the variables.

A *floating code allowing t times of rewriting* is a code that allows the variables to be rewritten at least t times in total, regardless of what the sequence of rewriting is. A fundamental objective of floating codes is to maximize t.

In the following, we first present a brief overview of the related work. Then, we present the constructions of several families of floating codes, which are either optimal or asymptotically optimal. We also present upper and lower bounds to t for general floating codes. The details are as follows.

II. Related Work

WOM codes were first studied by Rivest, Shamir [9] *et al.*, where a single variable is stored in a WOM and needs to be updated multiple times. Capacities of WOM have been studied [3] [4] [6] [7] [9] [10], and multiple classes of WOM codes have been invented. The majority of the known codes are binary, and they include linear codes [2] [9], tabular codes [9], codes constructed using Golay codes [2] or projective geometries [8], etc. Besides WOM, constrained memories also include write efficient memory (WEM), write unidirectional memory (WUM) and write isolated memory (WIM) [7].

There is no work we are aware of that addresses the use of codes for flash memories for increasing the number of (re)writes between two erasing operations, useful for improving writing speed and prolonging the memory lifetime. The use of error-correcting codes for improving data reliability in flash memories has been proposed in some works [1] [5].

III. An Optimal Code for Two Binary Variables

In this section, we present a floating code for binary variables. That is, $l = 2$, so each variable has value 0 or 1. In flash memories, the 16 bits of a word are usually stored at the same position of 16 parallel blocks. Consequently, a rewriting operation on a word becomes the rewriting of a bit in a block. Therefore, it is important to study the case of $l = 2$.

The code we present is for $k = 2, l = 2$ and arbitrary n and q. The code maximizes t, the number of rewrites, and is thus optimal. We prove the code's optimality by providing a general upper bound to t for floating codes, not limited to the case $k = 2, l = 2$.

A. Optimal Floating Code for $k = 2, l = 2$ and Arbitrary n, q

Three examples of the code are shown in Fig. 1, corresponding to $n = 1, 2$ and 3, respectively. We comment that $n = 1, 2$ are, in fact, degenerated cases; it is only when $n = 3$ or more that the code reveals the full structure of its construction.

The numbers inside each circle are a cell state vector, while the bold numbers beside the circle are the corresponding variable vector. For example, in Fig. 1(a), the cell state vector $(c_1) = (3)$ corresponds to the variable vector $(v_1, v_2) = (0, 0)$; in Fig. 1(c), the cell state vector $(c_1, c_2, c_3) = (1, 0, 0)$ corresponds to the variable vector $(v_1, v_2) = (1, 0)$. The arrows leaving a cell state vector shows how the next rewriting should be performed when this cell state vector is the current cell state vector. For example, for the code in Fig. 1(c), if the current cell state vector is $(1, 0, 0)$ and the new rewriting request is to change the first variable to '0' (which means to change the variable vector from $(1, 0)$ to $(0, 0)$), then the cell state vector will become $(1, 1, 0)$. Similarly, if the sequence of rewriting changes the variable vector as $(0, 0) \rightarrow (1, 0) \rightarrow (1, 1) \rightarrow (0, 1) \rightarrow \cdots$ (note that every rewriting changes the value of just one variable), the cell state vector changes as $(0, 0, 0) \rightarrow (1, 0, 0) \rightarrow (1, 0, 1) \rightarrow (1, 0, 2) \rightarrow \cdots$

We define the cell state vectors of the *i-th generation* to be the cell state vectors reachable after i times of rewriting. In Fig. 1, all the cell state vectors in the same generation are placed at the same horizontal level. For example, in Fig. 1(c), the cell state vectors in the 2nd generation are $(1, 1, 0), (1, 0, 1)$ and $(0, 1, 1)$. The codes in Fig. 1 are all for $q \rightarrow \infty$, and they all have periodic patterns; specifically, every code is a repetition of the structure shown in the dotted box labelled by "one period." To see how, notice that the first generation in the dotted box contains two cell state vectors corresponding to two different variable vectors, and so is true for the generation of cell state vectors directly following the dotted box; what's more, the latter two cell state vectors can be obtained from the former two cell state vectors by raising every cell's state by 2. (For example, in Fig. 1(b), the former two cell state vectors are $(1, 0)$ and $(0, 1)$; when we raise every cell's state by 2,

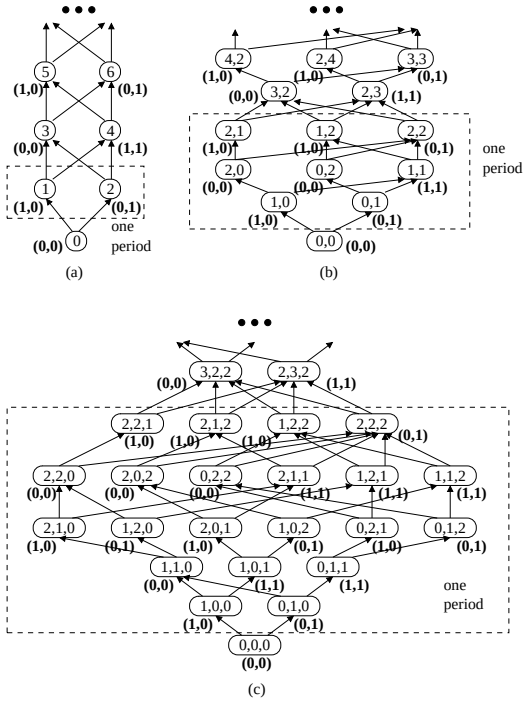

Fig. 1. Three examples of an optimal floating code for $k = 2, l = 2$ and arbitrary n, q. (a) $n = 1$. (b) $n = 2$. (c) $n = 3$.

we get $(3, 2)$ and $(2, 3)$, the latter two cell state vectors.) The code is built for arbitrarily large q in the following way. A "period" in the code contains $2n - 1$ generations. The second period directly follows – and has the same structure as – the first period, except that: (i) every cell's state is raised by 2, (ii) the pair of variable vectors $(1, 0)$ and $(0, 0)$ are switched, and the pair of variable vectors $(0, 1)$ and $(1, 1)$ are also switched. For $i = 1, 2, 3, \cdots$, the $(2i + 1)$-th (resp., $(2i + 2)$-th) period has the same structure as the 1st (resp. 2nd) period except that every cell's state is raised by $4i$.

If q is finite, it is simple to get the corresponding code: just truncate the above code to the maximum generation, subject to the constraint that every cell's state is at most $q - 1$.

We present the formal construction of the code in Fig. 2. The construction is in fact quite regular and elegant.

It is straightforward to verify the correctness (validity) of the code in Fig. 2. The key step is to verify that for every cell state vector, its two outgoing arrows enter two cell state vectors in the next generation that correspond to two different and correct variable vectors (v_1, v_2). It is also straightforward to verify the correctness of the following theorem.

Theorem 1: For the code constructed in Fig. 2, $t = (n - 1)(q - 1) + \lfloor \frac{q-1}{2} \rfloor$.

We see that the floating code integrates the WAM's rewriting capabilities for different variable to a very high degree. Let's call $\sum_{i=1}^{n} c_i$ the *weight* of the cell state vector. Clearly, every rewriting needs to increase that *weight* by at least 1. If the n cells are evenly split to be used independently by the $k = 2$ variables, t can never exceed $\frac{n}{2} \cdot (q - 1)$. The floating code, however, achieves $t \approx (n - 0.5)(q - 1)$.

1. For $i = 1, 2, \cdots, n - 1$, do:
   The i-th generation of cell state vectors contains all the $i + 1$ elements that satisfy the following properties: among the first $i+1$ cells, i of them are in state 1 and one of them is in state 0; the last $n - (i + 1)$ cells are all in state 0.
   In the i-th generation, if a cell state vector is $(1, 1, \cdots, 1, 0, 0, \cdots, 0)$ (that is, the first i cells are in state 0, and the last $n-i$ cells are in state 0), then it corresponds to the variable vector $(v_1, v_2) = (1, 0)$ (if i is odd) or $(v_1, v_2) = (0, 0)$ (if i is even); otherwise, the cell state vector corresponds to the variable vector $(v_1, v_2) = (0, 1)$ (if i is odd) or $(v_1, v_2) = (1, 1)$ (if i is even).
   Let a denote a cell state vector in the $(i - 1)$-th generation. The two outgoing arrows of a are as follows: one arrow goes to the cell state vector in the i-th generation where the first i cells are in state 1 and the last $n - i$ cells are in state 0; the other arrow goes to the cell state vector of the i-th generation that is the same as a except that its $(i + 1)$-th cell is in state 1 instead of state 0.
2. Note that by the above construction, the $(n - 1)$-th generation contains n cell state vectors, where each cell state vector has $n-1$ cells in the state 1 and one cell in the state 0. Let's denote those n cell state vectors by $s_1, s_2, \cdots, s_n$. For s_i ($1 \le i \le n$), let's denote the $n-1$ cells in state 1 by $b_{\pi(i,1)}, b_{\pi(i,2)}, \cdots, b_{\pi(i,n-1)}$, and denote the cell in state 0 by $b_{\pi(i,n)}$. ($1 \le \pi(i, j) \le n$.)
3. For $i = n, n + 1, \cdots, 2n - 3$, do:
   The i-th generation of cell state vectors contains $n(i - n + 2)$ elements, which we partition into n groups. For $j = 1, 2, \cdots, n$, the j-th group contains all the $i-n+2 = [i-(n-1)]+1$ elements that satisfy the following properties: among the $[i - (n - 1)] + 1$ cells $b_{\pi(j,1)}, b_{\pi(j,2)}, \cdots, b_{\pi(j,i-(n-1)+1)}$, $i - (n - 1)$ of them are in state 2 and one of them is in state 1; the $2n - i - 3$ cells $b_{\pi(j,i-(n-1)+2)}, b_{\pi(j,i-(n-1)+3)}, \cdots, b_{\pi(j,n-1)}$ are all in state 1; the cell $b_{\pi(j,n)}$ is in state 0.
   In the i-th generation, for a cell state vector in the j-th group, if the cells $b_{\pi(j,1)}, b_{\pi(j,2)}, \cdots, b_{\pi(j,i-(n-1))}$ are all in state 2, then it corresponds to the variable vector $(v_1, v_2) = (1, 0)$ (if i is odd) or $(v_1, v_2) = (0, 0)$ (if i is even); otherwise, the cell state vector corresponds to the variable vector $(v_1, v_2) = (0, 1)$ (if i is odd) or $(v_1, v_2) = (1, 1)$ (if i is even).
   Let a denote a cell state vector in the $(i - 1)$-th generation and in the j-th group. (If $i - 1 = n - 1$, then let a be s_j.) The two outgoing arrows of a are as follows: one arrow goes to the cell state vector in the i-th generation and the j-th group where the $i - (n - 1)$ cells $b_{\pi(j,1)}, b_{\pi(j,2)}, \cdots, b_{\pi(j,i-(n-1))}$ are all in state 2; the other arrow goes to the cell state vector in the i-th generation and the j-th group that is the same as a except that its cell $b_{\pi(j,i-(n-1)+1)}$ is in state 2 instead of state 1.
4. Note that by the above construction, the $(2n - 3)$-th generation contains $n(n - 1)$ cell state vectors, where each vector has $n - 2$ cells in state 2, one cell in state 1, and one cell in state 0.
5. The $(2n - 2)$-th generation of cell state vectors contains $n + \binom{n}{2}$ elements, which we partition into two groups. The first group contains all the n vectors where $n - 1$ cells are in state 2 and one cell is in state 0. The second group contains all the $\binom{n}{2}$ vectors where $n - 2$ cells are in state 2 and two cells are in state 1. All the cell state vectors in the first (resp., second) group correspond to the variable vector $(v_1, v_2) = (0, 0)$ (resp., $(1, 1)$).
   The $(2n - 1)$-th generation of cell state vectors contains $n + 1$ elements, which we partition into two groups. The first group contains all the n cell state vectors where $n-1$ cells are in state 2 and one cell is in state 1; the second group contains one cell state vector where all the n cells are in state 2. The cell state vectors in the first (resp. second) group correspond to the variable vector $(v_1, v_2) = (1, 0)$ (resp. $(0, 1)$).
   Let a denote a cell state vector in the $(2n - 3)$-th (resp., $(2n - 2)$-th) generation. The two outgoing arrows of a enter two cell state vectors of the $(2n - 2)$-th (resp., $(2n - 1)$-th) generation, respectively in the first group and in the second group, both of which are *above* a.
6. The above $2n - 1$ generations of cell state vectors form the first *period* of the code. Repeat the period's structure to get the 2nd, 3rd, $\cdots$ periods (as described before in this paper). Just remember that for the i-th period, if i is even, then switch the variable vector $(0, 0)$ with $(1, 0)$, and switch the variable vector $(1, 1)$ with $(0, 1)$. If q is finite, truncate the code to the maximum generation subject to the constraint that all the cells' states are at most $q - 1$.

Fig. 2. Construction of a code for $k = 2, l = 2$ and arbitrary n, q.

The code construction in Fig. 2 can be easily converted into very efficient encoding (for rewriting) and decoding (for mapping cell state vectors to variable vectors) algorithms. Due to the space limitation, we skip the details.

B. A General and Tight Upper Bound to t

We now present a general upper bound to t, which holds for any k, l, n and q. The bound can show that the code in Fig. 2 is optimal.

Theorem 2: 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 $n \geq k(l-1)-1$.

Assume that the floating code is given. Note that since a rewriting operation can update any of the k variables, and every variable has $l-1$ possible values that are different from its current value, a rewriting operation has $k(l-1)$ possibilities that do change the value of the variables. We will choose a sequence of rewriting operations $W_1, W_2, W_3\cdots$, where W_i denotes the i-th rewriting operation. For $i = 1, 2, \cdots, n$ and $j = 0, 1, 2, \cdots$, let a_i denote the i-th cell, and let c_i^j denote the state of the i-th cell after the j-th rewriting operation. (So $0 \leq c_i^j \leq q-1$, and $c_i^0 = 0$.) We choose the sequence of rewriting operations – and build a sequence of sets $S_0, S_1, S_2, \cdots$ at the same time – in the following way:

1. Let S_0 be any subset of $\{a_1, a_2, \cdots, a_n\}$ of cardinality $k(l-1)-1$. (For example, we can let $S_0 = \{a_1, a_2, \cdots, a_{k(l-1)-1}\}$.)
2. For $i = 1, 2, 3, \cdots$, do:
   { For the i-th rewriting operation W_i, if all the $k(l-1)$ possible choices increase the *weight of the cell state vector* (defined as $\sum_{i=1}^{n} c_i$, as before) by 1, choose W_i to be a rewriting operation that increases the state of a cell $p \notin S_{i-1}$ by 1; otherwise, choose W_i to be a rewriting operation that increases the *weight of the cell state vector* by at least 2.
   Let S_i be a subset of $\{a_1, a_2, \cdots, a_n\}$ of cardinality $k(l-1)-1$ satisfying this property: for any two cells $a_x \in S_i$ and $a_y \notin S_i$, $c_x^i \leq c_y^i$. (In other words, S_i contains $k(l-1)-1$ cells whose states are lower than or equal to the others' after i rewriting operations.) }

We use the above method to keep obtaining rewriting operations until no more rewriting is allowed by the floating code. Say that the above method gives us totally t_0 rewriting operations: $W_1, W_2, \cdots, W_{t_0}$. We will prove that $t_0 \leq [n-k(l-1)+1]\cdot(q-1)+\lfloor\frac{[k(l-1)-1]\cdot(q-1)}{2}\rfloor$, which will in turn prove the final conclusion.

For $i = 0, 1, \cdots, t_0$, let $P_i = \sum_{a_j \in S_i}(q-1-c_j^i)$, and let $Q_i = \sum_{a_j \notin S_i}(q-1-c_j^i)$. We now use induction to prove the following *assertion*:

- *Assertion:* For any $0 \leq i \leq t_0$, $t_0 - i \leq Q_i + \lfloor\frac{P_i}{2}\rfloor$. (Note that $t_0 - i$ is the number of rewriting operations we have after the i-th rewriting operation.)

The induction is in the reverse order of the rewriting operations. When $i = t_0$, the *assertion* is true because $t_0 - i = 0$ and $Q_i \geq 0$, $P_i \geq 0$. That is our base case. Now we start the induction.

Assume that the *assertion* holds for any $i > I_0$. Now consider the case $i = I_0$. There are two subcases:

Subcase 1: The (I_0+1)-th rewriting operation, W_{I_0+1}, increases the weight of the cell state vector *by 1.* Then, let a_x denote the cell whose status is raised by 1 by W_{I_0+1}. It is not difficult to see that $a_x \notin S_{I_0}$, $a_x \notin S_{I_0+1}$, $c_x^{I_0+1} = c_x^{I_0} + 1$, $c_j^{I_0+1} = c_j^{I_0}$ for any $j \neq x$, $P_{I_0} = P_{I_0+1}$, and $Q_{I_0} = Q_{I_0+1}+1$. By the induction assumption, $t_0-(I_0+1) \leq Q_{I_0+1} + \lfloor\frac{P_{I_0+1}}{2}\rfloor$, so we get $t_0 - I_0 \leq Q_{I_0} + \lfloor\frac{P_{I_0}}{2}\rfloor$. So the *assertion* holds.

Subcase 2: The (I_0+1)-th rewriting operation, W_{I_0+1}, increases the weight of the cell state vector *by at least 2.* Let $P'_{I_0} = \sum_{a_j \in S_{I_0}}(q-1-c_j^{I_0+1})$, and let $Q'_{I_0} = \sum_{a_j \notin S_{I_0}}(q-1-c_j^{I_0+1})$. Since W_{I_0+1} increases the *weight of the cell state vector* by at least 2, there are two possibilities; (1) $Q_{I_0} \geq Q'_{I_0}+1$ and $P_{I_0} \geq P'_{I_0}$; (2) $Q_{I_0} = Q'_{I_0}$ and $P_{I_0} \geq P'_{I_0}+2$. In both cases, we get $Q_{I_0} + \lfloor\frac{P_{I_0}}{2}\rfloor \geq 1 + Q'_{I_0} + \lfloor\frac{P'_{I_0}}{2}\rfloor$.

Now we compare $Q'_{I_0} + \lfloor\frac{P'_{I_0}}{2}\rfloor$ with $Q_{I_0+1} + \lfloor\frac{P_{I_0+1}}{2}\rfloor$. Let's partition the n cells into four sets A, B, C, D as follows: $A = \{a_j | a_j \in S_{I_0}, a_j \in S_{I_0+1}\}$, $B = \{a_j | a_j \notin S_{I_0}, a_j \notin S_{I_0+1}\}$, $C = \{a_j | a_j \in S_{I_0}, a_j \notin S_{I_0+1}\}$, $D = \{a_j | a_j \notin S_{I_0}, a_j \in S_{I_0+1}\}$. By definition, $P'_{I_0}, Q'_{I_0}, P_{I_0+1}, Q_{I_0+1}$ are all summations of terms of the form $q-1-c_x^y$; we see the value of $q-1-c_x^y$ as the *contribution of the cell a_x*. It is not difficult to see that every cell in A or B contributes the same value to $Q'_{I_0} + \lfloor\frac{P'_{I_0}}{2}\rfloor$ and $Q_{I_0+1} + \lfloor\frac{P_{I_0+1}}{2}\rfloor$. As for the cells in C and D, since $|S_{I_0}| = |S_{I_0+1}| = k(l-1)-1$, $|C| = |D|$. So we can partition the cells in $C \cup D$ into pairs in the form of $(a_x \in C, a_y \in D)$. Consider a pair $(a_x \in C, a_y \in D)$. Clearly, $c_x^{I_0+1} \geq c_y^{I_0+1}$ because $a_x \notin S_{I_0+1}$ and $a_y \in S_{I_0+1}$. So a_x and a_y together contribute more to $Q'_{I_0} + \lfloor\frac{P'_{I_0}}{2}\rfloor$ than to $Q_{I_0+1} + \lfloor\frac{P_{I_0+1}}{2}\rfloor$. By combining the above results, we get $Q'_{I_0} + \lfloor\frac{P'_{I_0}}{2}\rfloor \geq Q_{I_0+1} + \lfloor\frac{P_{I_0+1}}{2}\rfloor$.

Therefore, in *Subcase 2*, $Q_{I_0} + \lfloor\frac{P_{I_0}}{2}\rfloor \geq 1 + Q_{I_0+1} + \lfloor\frac{P_{I_0+1}}{2}\rfloor$. By the induction assumption, $t_0-(I_0+1) \leq Q_{I_0+1} + \lfloor\frac{P_{I_0+1}}{2}\rfloor$, so we get $t_0 - I_0 \leq Q_{I_0} + \lfloor\frac{P_{I_0}}{2}\rfloor$. So the *assertion* again holds. So in any case, the *assertion* holds when $i = I_0$.

We have now proved by induction that the *assertion* holds for all $0 \leq i \leq t_0$. Note that $P_0 = |S_0|(q-1) = [k(l-1)-1](q-1)$, and $Q_0 = (n-|S_0|)(q-1) = [n-k(l-1)+1](q-1)$. By making $i = 0$ in the assertion, we get $t_0 \leq [n-k(l-1)+1]\cdot(q-1)+\lfloor\frac{[k(l-1)-1]\cdot(q-1)}{2}\rfloor$. Since $W_1, W_2, \cdots, W_{t_0}$ is a maximal sequence of writing operations, $t \leq t_0$. So for any floating code, $t \leq [n-k(l-1)+1]\cdot(q-1)+\lfloor\frac{[k(l-1)-1]\cdot(q-1)}{2}\rfloor$.

Now, consider the case $n < k(l-1)-1$. There are $k(l-1)$ possible choices for a rewriting operation (that changes the value of the variables), but there are only $n < k(l-1)$ cells. So there is always a choice for the next rewriting operation that can increase the *weight of the cell state vector* by at least 2. We choose a maximal sequence of rewriting operations $W_1, W_2, \cdots, W_{t_0}$ such that every W_i increases the

weight of the cell state vector by at least 2. The *weight of the cell state vector* can never exceed $n(q-1)$. So $t_0 \leq \lfloor \frac{n(q-1)}{2} \rfloor$. So $t \leq \lfloor \frac{n(q-1)}{2} \rfloor$. ∎

Theorem 2 shows that the code we presented in Fig. 2 is optimal. To see why, just make $k = 2$ and $l = 2$ in Theorem 2, and compare it with Theorem 1. Therefore,

Theorem 3: The floating code presented in Fig. 2 is optimal, namely, it maximizes the number of times of rewriting t.

The above observation also shows that whenever $k = 2$ and $l = 2$, the upper bound shown in Theorem 2 is exact. In this sense, the bound is tight.

IV. Asymptotically Optimal Linear Codes

In this section, we present two linear codes for binary variables. Both codes have $\lim_{n\to\infty} t = (q-1)n + o(n)$. Since all floating codes have $t \leq (q-1)n$, the two codes are asymptotically optimal in n.

In both codes, every cell essentially corresponds to an integer, and a linear combination of those integers form the numerical representation of the k variables. We borrow the idea from the WOM codes proposed by Fiat and Shamir in [3]. Those WOM codes are for updating a single variable in a binary WOM. The floating codes we present are, respectively, for rewriting two or three variables in q-ary WAMs.

We define a function $odd(x)$ as follows: for any non-negative integer x, if x is odd, $odd(x) = 1$; otherwise, $odd(x) = 0$. Let $(a_1^{x_1} a_2^{x_2} \cdots a_h^{x_h})$ denote a string that consists of x_1 consecutive a_1's, followed by x_2 consecutive a_2's, $\cdots$, ended with x_h consecutive a_h's. For example, $(1^2 0^1 1^1 0^3)$ is $(1, 1, 0, 1, 0, 0, 0)$. Given the value of k binary variables $x = (v_1, v_2, \cdots, v_k)$, $f(x)$ maps x to a number between 0 and $2^k - 1$: $f(x) = v_1 \cdot 2^{k-1} + v_2 \cdot 2^{k-2} + \cdots + v_k \cdot 2^0$.

Below are the constructions of the two floating codes.

- *Code Construction I: $k = 2, l = 2, n \geq 3$, arbitrary q*
  In this code, a valid cell state vector $(c_1, c_2, \cdots, c_n)$ always satisfies the following two constraints: (1) $\forall\ i, j$, $|c_i - c_j| \leq 1$; (2) $(c_1, c_2, \cdots, c_n) = ((a+1)^{x_1} a^{x_2} (a+1)^{x_3})$ for some a, x_1, x_2, x_3 where $0 \leq a < q-1, x_1 + x_2 + x_3 = n, x_2 \geq 1$. (For example, when $n = 5, q = 3$, $(1, 1, 1, 0, 1) = (1^3 0^1 1^1)$ and $(1, 1, 1, 1, 2) = (2^0 1^4 2^1)$ are both valid cell state vectors.)
  A cell state vector $((a+1)^{x_1} a^{x_2} (a+1)^{x_3})$ corresponds to the variable vector $y = (v_1, v_2, \cdots, v_k)$ in the following way: $f(y) = odd(x_1) \cdot 2 + odd(x_3)$. (For example, when $n = 5, q = 3$, both cell state vectors $(1, 0, 0, 0, 1)$ and $(2, 2, 2, 1, 2)$ correspond to the variable vector $(v_1, v_2) = (1, 1)$.)
  The rewriting operation is as follows. When the rewriting changes the value of variable v_1 (resp., v_2), we usually increase x_1 (resp., x_3) by 1 and decrease x_2 by 1. The exception happens when $x_2 = 1$; in that case, we first raise all the cells to the state $a + 1$ (which makes $x_1 = x_3 = 0$ and $x_2 = n$), then increase x_1 or x_3 (or both) based on necessity.
  For example, assume that $n = 4, q = 3$ and the rewriting operations change the variable vector (v_1, v_2) as follows: $(0, 0) \to (0, 1) \to (0, 0) \to (1, 0) \to (1, 1)$. Then, the cell state vector changes as follows: $(0, 0, 0, 0) \to (0, 0, 0, 1) \to (0, 0, 1, 1) \to (1, 0, 1, 1) \to (2, 1, 1, 2)$.

Theorem 4: When n is odd, the floating code in Code Construction I has $t = (n-1)(q-1)$; if n is even, it has $t = (n-2)(q-1) + 1$.

Proof: When every cell is either in state a or $a+1$, we say that the cells are in *phase* $a + 1$. So the rewriting operations change the cells from phase 1 to phase 2 to $\cdots$ to phase $q-1$. Consider phase 1. Every rewriting increases x_1 or x_3 by 1, and $x_1 + x_3 \leq n - 1$. So $n - 1$ rewriting operations can happen in phase 1.

When a rewriting operation changes the cells from phase 1 to phase 2, the following analysis considers the worst cases: (1) When n is even, the rewriting operation can make the variable vector become $(v_1, v_2) = (1, 1)$, so in phase 2, both x_1 and x_3 need to be set as 1; (2) When n is odd, the rewriting operation cannot make the variable vector be $(1, 1)$. That is because right before the rewriting operation, $x_1 + x_3 = n - 1$ is even, so the variable vector is either $(0, 0)$ or $(1, 1)$. So the rewriting operation changes the variable vector to be either $(1, 0)$ or $(0, 1)$; therefore in phase 2, either $x_1 = 1, x_3 = 0$ or $x_1 = 0, x_3 = 1$. So when n is odd (resp., even), $n - 1$ (resp., $n - 2$) rewriting operations can happen in phase 2. Phases $3, 4, \cdots, q-1$ are the same as phase 2. That leads to the final conclusion. ∎

By theorems 2 and 4, we see that when $q = 2$, the above code is strictly optimal.

- *Code Construction II: $k = 3, l = 2, n \geq 5$, arbitrary q*
  In this code, a valid cell state vector $(c_1, c_2, \cdots, c_n)$ always satisfies the following two constraints: (1) $\forall\ i, j$, $|c_i - c_j| \leq 1$; (2) the cell state vector is either in the form $((a+1)^{x_1} a^{x_2} (a+1)^{x_3} a^{x_4} (a+1)^{x_5})$, where $\sum_{i=1}^{5} x_i = n, x_2 \geq 1, x_4 \geq 1$ (which we call *form I*), or in the form $((a+1)^{x_1} a^{x_2} (a+1)^{x_5})$, where $x_1 + x_2 + x_5 = n, x_2 \geq 1$ (which we call *form II*).
  A cell state vector corresponds to the variable vector $y = (v_1, v_2, \cdots, v_k)$ in the following way: if the cell state vector is in *form I*, then $f(y) = odd(x_1) \cdot 4 + odd(x_3) \cdot 2 + odd(x_5)$; if the cell state vector is in *form II*, then $f(y) = odd(x_1) \cdot 4 + odd(x_5)$.
  The rewriting operation is as follows. When the rewriting changes the value of variable v_1 (resp., v_3), we usually increase x_1 (resp., x_5) by 1 and decrease x_2 (resp., x_4 or x_2, depending on if the cell state vector is in *form I* or *form II*) by 1. When the rewriting changes the value of variable v_2, we either increase x_3 by 1 and decrease x_2 or x_4 by 1 (when the cell state vector is in *form I*), or change the cell in the middle of the sequence of a's from state a to state $a+1$ (when the cell state vector is in *form II*). If x_2 or x_4 becomes zero due to the above operation, the cell state vector is reevaluated, and the operation described above is carried out again based on the values of the

variables. If the above operation cannot be carried out any more when the cells remain in the current two states – state a and state $a+1$ – then we start to use the two states $a+1$ and $a+2$, in the same way as we have used the two states a and $a+1$ above.

The following examples show how the code works. Assume that $n = 10, q = 3$. (1) If the cell state vector is $(1^1 0^9 1^0)$, then $(v_1, v_2, v_3) = (1, 0, 0)$; if the next two rewriting operations change (v_1, v_2, v_3) to $(0, 0, 0)$ and then to $(0, 1, 0)$, the cell state vector changes to $(1^2 0^8 1^0)$, and then to $(1^2 0^3 1^1 0^4 1^0)$. (2) If the cell state vector is $(1^3 0^1 1^1 0^4 1^1)$, then $(v_1, v_2, v_3) = (1, 1, 1)$; if the next two rewriting operations change (v_1, v_2, v_3) to $(0, 1, 1)$ and then to $(0, 1, 0)$, the cell state vector changes to $(1^6 0^1 1^1 0^1 1^1)$, and then to $(2^0 1^4 2^1 1^5 2^0)$.

Theorem 5: The floating code in Code Construction II has $t \geq (n - 6 - 2\log_2 n)(q-1) + 2$.

Proof: A rewriting operation increases the *weight of the cell state vector* by one except in the following three occasions: (1) The rewriting makes x_2 or x_4 become zero, in which case the *weight of the cell state vector* can be increased by at most 3; (2) The rewriting cannot be accomplished while the cells continue to use the current two states, which happens only if $x_2 + x_4 \leq 4$ before the rewriting; (3) When the previous case happens, the rewriting is accomplished by making cells use the next pair of states, which leads to $x_1 \leq 1, x_3 \leq 1, x_5 \leq 1$ (that is, increasing the *weight of the cell state vector* by at most 3).

Let a and $a+1$ indicate the two states that the cells are in. Every time after the cell state vector changes from *form II* into *from I*, case (1) can happen only once. For any fixed a, the cell state vector can change into *form I* no more than $\log_2 n$ times, because such a change is caused by splitting a sequence of consecutive cells in state a into two nearly equally long subsequences in state a – with a cell of state $a+1$ separating them – and the length of this sequence at least halves every time. So case (1) happens at most $(q-1)\log_2 n$ time in total. Both case (2) and case (3) happen at most once for any fixed a, and case (3) happens only if $a > 0$. Therefore, if we use z_1, z_2, z_3 to represent, respectively, the numbers of times that cases (1), (2) and (3) happen, and use z_4 to represent the number of rewriting operations that do not involve those three cases, then $3z_1 + 4z_2 + 3z_3 + z_4 \geq n(q-1)$. Since $z_1 \leq (q-1)\log_2 n, z_2 \leq q-1, z_3 \leq q-2$, we get $z_1 + z_3 + z_4 \geq (n-6-2\log_2 n)(q-1)+2$. So $t \geq (n-6-2\log_2 n)(q-1)+2$. ■

V. Bounds for Floating Codes

A general upper bound to t has been shown in theorem 2. It has also been shown that when $k = 2, l = 2$, the bound is exact. For large k or l, the following theorem can give a better upper bound.

Theorem 6: Let w be the smallest positive integer such that $\binom{w+n}{n} \geq l^k$. Then, $t \leq \lceil \frac{(q-1)n}{w} \rceil k$.

Let w' be the smallest positive integer such that $\binom{w'+n}{n} > l^k$. Then, when $k \geq 2$, $t \leq \lceil \frac{(q-1)n}{w'} \rceil k$.

Proof: First, consider the general case $k \geq 1$. Define S as $S = \{(a_1, a_2, \cdots, a_n) | \sum_{i=1}^n a_i \leq w, a_1, a_2, \cdots, a_n$ are non-negative integers$\}$, and let w be the smallest integer such that $|S| \geq l^k$. Define S' as $S' = \{(d_1, d_2, \cdots, d_n) | \sum_{i=1}^n d_i \leq w+n, d_1, d_2, \cdots, d_n$ are positive integers$\}$. By letting $d_i = a_i + 1$ for $i = 1, 2, \cdots, n$, we see that there is a one-to-one mapping between S and S'. So $|S| = |S'|$. An element $(d_1, d_2, \cdots, d_n)$ belongs to S' if and only if it is a solution to the following problem: partition a path of $w+n$ vertices into n or more sub-paths such that for $i = 1, 2, \cdots, n$, the i-th sub-path has $d_i > 0$ vertices. Therefore, $|S'| = \binom{w+n}{n}$. So w is also the smallest positive integer such that $\binom{w+n}{n} \geq l^k$.

k consecutive rewriting operations can make the variables change to or go through any of the l^k possible values. If we see a_i (for $i = 1, 2, \cdots, n$) as the increase in c_i – the state of the i-th cell – and consider the way S and w are defined, we see that whatever the current cell state vector is, there exist k consecutive rewriting operations that increases the *weight of the cell state vector* $\sum_{i=1}^n c_i$ by at least w. Now consider the first batch of such k rewriting operations, the second batch, and so on. Since the maximum weight of the cell state vector is $(q-1)n$, we get $t \leq \lceil \frac{(q-1)n}{w} \rceil k$.

For the slightly more restrictive case $k \geq 2$, we refine the above proof a little. When $k \geq 2$, among the cell state vectors that k consecutive rewriting operations can make the cell state vector change to or go through, there are at least two cell state vectors (including the current cell state vector) that correspond to the current variable vector. The rest of the proof is similar. ■

When k or l is sufficiently large, theorem 6 gives an upper bound to t that is roughly $\frac{(q-1)nk}{(n!)^{\frac{1}{n}} l^{\frac{k}{n}}}$. Now we present an elementary lower bound.

Theorem 7: There exist floating codes where $t \geq \lfloor \frac{n}{k} \rfloor \cdot \lfloor \frac{q-2}{l-1} \rfloor$.

Proof: We show a code that achieves the bound. For $i = 1, 2, \cdots, k$ and $j = 0, 1, \cdots, \lfloor \frac{n}{k} \rfloor - 1$, let the $(i+jk)$-th cell be used for the i-th variable. The i-th variable is first encoded using the i-th cell, then the $(i+k)$-th cell, and so on. For a cell, the value of the variable it corresponds to equals its state modulo l unless its state becomes $q-1$, which indicates that this cell is "no longer usable." This gives a code with $t = \lfloor \frac{n}{k} \rfloor \cdot \lfloor \frac{q-2}{l-1} \rfloor$. ■

Theorem 8: When k, l, q are fixed and $n \to \infty$, there exist floating codes where $t = (q-1)n + o(n)$.

Proof: The idea is on the conversion between floating codes and WOM codes. A (binary) WOM is a special case of WAM with $q = 2$, and a WOM code is a special case of floating codes with $k = 1$. Rivest and Shamir have shown a tabular WOM code [9] achieving $t = n + o(n)$ as l is fixed. We can see the k variables from an alphabet of size l as a super variable from an alphabet of size l^k. Then, every rewriting for the k variables is an instance of the rewriting of the super variable (although not *vice versa*). We can therefore use the WAM layer by layer: first use the states 0 and 1 as much as possible, then use the states 1 and 2 in the same way, $\cdots$, then

use the states $q-2$ and $q-1$. For each layer, apply the tabular WOM code to the super variable. That gives us a floating code with $t = (q-1)n + o(n)$. ■

Theorem 8 shows that when $n \to \infty$, floating codes can integrate the WAM's rewriting capabilities for different variables nearly perfectly.

VI. Conclusions

Floating codes for WAMs have been explored in this paper. Both optimal/asymptotically optimal floating codes and performance bounds for general codes have been presented. They show that floating codes can integrate very well the rewriting capabilities of different variables in many cases. Such an ability is useful for the storage of multiple variables in WAMs, including flash memories, etc. as example applications. We will continue the exploration of floating codes, and expand the knowledge on coding in memories with irreversible state transitions.

References

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

[2] G. D. Cohen, P. Godlewski and F. Merkx, " Linear binary code for write-once memories," *IEEE Trans. Inform. Theory*, vol. IT-32, pp. 697-700, Sept. 1986.

[3] A. Fiat and A. Shamir, "Generalized 'write-once' memories," *IEEE Trans. Inform. Theory*, vol. IT-30, pp. 470-480, May 1984.

[4] 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.

[5] S. Gregori, A. Cabrini, O. Khouri and G. Torelli, "On-chip error correcting techniques for new-generation flash memories," *Proceedings of The IEEE*, vol. 91, no. 4, April 2003.

[6] C. Heegard, "On the capacity of permanent memory," *IEEE Trans. Inform. Theory*, vol. IT-31, pp. 34-42, Jan. 1985.

[7] A. V. Kuznetsov and A. J. H. Vinck, "On the general defective channel with informed encoder and capacities of some constrained memories," *IEEE Trans. Inform. Theory*, vol. 40, no. 6, pp. 1866-1871, Nov. 1994.

[8] F. Merkx, "WOMcodes constructed with projective geometries," *Traitement du Signal*, vol. 1, no. 2-2, pp. 227-231, 1984.

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

[10] 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.