Combining the Phoenix Flash Code with the Binary Index Flash Code for Low Write Deficiency

G. N. Corneby[1], L. K. Sanchez[1], P. Fernandez[1], M. J. Tan[2], and K. Kaji[2]
[1]Department of Information Systems and Computer Science, Ateneo de Manila University, Philippines
[2]Graduate School of Information Science, Nara Institute of Science and Technology, Japan

Abstract— *In the framework of floating codes, a block of flash cells stores data in the form of binary numbers. The fundamental approach in constructing a coding scheme is by assigning cells to bits. However, the way to assign cells to bits is not simple, as the frequency of changes of the value of the bits is not known. This makes it difficult to partition the cells in a block in proportion to the frequency of the changes of bit value where the most updated bit has the most number of cells assigned to it. In this study, we discuss a novel coding scheme to dynamically assign cells to bits. At the beginning, there is a pre-determined assignment of cells to bits, but the coding scheme allows reassignment of cells if needed. The proposed coding scheme gives a very low write deficiency. A novel idea is discussed in this manuscript.*

Keywords: flash code, flash memory, phoenix flash code, binary-indexed, absorb, revive

1. Introduction

Flash memory devices are currently constrained by the write asymmetry property. It is easy to increase the charge in one flash cell, i.e., perform a *cell write*, but decreasing a charge in a cell is not possible except by emptying the charges simultaneously in all cells of a block. This operation is referred to as a *block erasure*.

A block erasure is not only time-consuming but also causes some damage to the device. It has been estimated that a block of cells can only accommodate about 10^4 to 10^5 block erasures before it becomes unreliable [5]. It is therefore desirable to delay block erasures as much as possible, by designing good coding schemes, in order to extend the lifespan of flash memory devices.

A *flash code* is used for decoding and encoding digital information in a flash memory. The performance of a flash code is normally evaluated by measuring its *write deficiency*. This is computed by taking the difference between the maximum possible and the actual number of cell writes. A lower write deficiency is clearly preferred.

One of the most popular flash codes in literature is the Index-less Indexed Flash Code (ILIFC). This flash code partitions a block of cells into sub-blocks, called *slices*. Each ILIFC slice has exactly k cells, where k is the number of bits of the data represented by a block. Encoding is designed so that it is possible to infer both the bit index (that a slice represents) and the bit-value by just reading the cell-values within the given slice [4]. Binary Index Flash Code (BIFC) introduced the partitioning of a flash code into smaller slices. Unlike ILIFC which uses k cells per slice, BIFC uses slices of size $s = O(\log k)$. The drawback in BIFC is that there is an overhead write deficiency of $s - 2$ for every slice. Generally, however, the BIFC flash code has a better write deficiency than the ILIFC when k is sufficiently large [8].

More recently, the Dual Mode Flash Code (DMFC) was introduced, combining the BIFC with a Simple Segmentation coding scheme. Experimental results using DMFC show that it has a significantly lower write deficiency than any of the previously designed flash codes in the average case [10].

In this study, we improve the DMFC further by using Phoenix Flash Code (PFC), a recently developed coding scheme that allows reassignment of cells to bits.[1] In the original version of PFC, the reassignment of cells contains a slight overhead cost which contribute to $O(n)$ to the write deficiency. We introduce a modification to the method to remove such overhead. The detailed discussion of PFC is in Sect. 3. Afterwards, we discuss combining PFC with BIFC in Sect. 4. Computer simulations show that this flash code is superior, even when compared to the DMFC and this is shown in Sect. 5.

2. Preliminaries

A *block* of flash memory is a sequence of n cells. Each cell stores an integer value from $A_q = \{0, \ldots, q-1\}$, and this value is referred to as the *cell-level*. A cell has three type states. A cell with a value of 0 or $q-1$ is said to be *empty* or *full*, respectively. If a cell is neither empty nor full, then that cell is said to be *active*. All cells within a block are ordered. The value of the i-th cell is denoted by c_i where $0 \leq i < n$. A tuple $(c_0, \ldots, c_{n-1}) \in A_q^n$ represents a possible state of a block. For two states $C = (c_0, \ldots, c_{n-1})$ and $C' = (c'_0, \ldots, c'_{n-1})$, we write $C \preceq C'$ if $c_i \preceq c'_i$ for all $0 \leq i < n$, and $C \prec C'$ if $C \preceq C'$ and $C \neq C'$. A state can transit from C to C' if and only if $C \prec C'$, as state transition is accomplished through cell writes. Similar to individual cells, a block has three type of states. A block is *empty* if all cells within the block are empty. A block is *full* if all the cells within a block are full. Otherwise, the block is *active*. The notion of "states", "$\prec$" and "type of

states" are extended to subsets of cells in a natural manner [2], [9]. A block of flash memory cells stores a k-bit data $D = (d_0, \ldots, d_{k-1})$. The data D is updated through a *write operation* which flips the value of a single bit in D.

A flash code $F = (\mathcal{E}, \mathcal{D})$ contains two functions which are used to update and retrieve the data stored within a block. The decode function $\mathcal{D} : A_q^n \to (d_0, \ldots, d_{k-1})$ retrieves the value of the data stored in the block of flash memory cells. The encode function $\mathcal{E} : \{0, \ldots, k-1\} \times A_q^n \to A_q^n \cup \{E\}$ is applied to the block for every write operation. The encode function first attempts to accommodate the write operation by applying cell writes to the block following a pre-specified procedure. If successful, this operation produces a new block state $C' = \mathcal{E}(i, C)$ where $\mathcal{D}(C)$ and $\mathcal{D}(C')$ only differ in the i-th bit. Otherwise, the encode function returns a block erasure E.

Since a block has n cells and each cell has a maximum of $q-1$ levels, then in the ideal case a flash code can accommodate each write operation with one cell write. Hence, the maximum number of write operations by the ideal flash code is $n(q-1)$. This expression is used in a metric for the performance of flash codes. Specifically, the *write deficiency* of a flash code F, denoted by $\delta(F)$, is computed using the formula $\delta(F) = n(q-1) - t$, where t is the actual number of write operations accommodated by the flash code F. A write deficiency of zero is the ideal case.

3. Phoenix Flash Code

Phoenix Flash Code (PFC) is a very recently developed flash code. Its encoding process is somewhat similar to *stacked segment encoding (SS encoding)*[10]. This is especially true when there is an equal distribution of write operations among the k bits of data. PFC starts with dividing the block into smaller groups called *segments*. A segment $S_i = (c_{i,0}, \ldots, c_{i,k-1})$ where $c_{i,j} = c_{ik+j}$, $0 \leq i < \frac{n}{k}$. Each segment has k cells and is cyclic in the context of the cell adjacency. This implies that the left adjacent cell of $c_{i,0}$ is $c_{i,k-1}$ and the reverse also holds true. For each active segment S_i, cell $c_{i,j}$ is initially assigned to d_j where $0 \leq j < k$. The bit-value of d_j is computed using the parity of the cell assigned to it. An example is shown in Fig. 1 using segment S_1.

Following the encoding process of the SS encoding, it can be observed that the assigning of cells to bits is not flexible enough to accommodate non-uniform distribution of write operations. PFC solves this problem by incorporating two operations, called *absorption* and *revival*. These two operations are used only when one cell is about to become full in the current write operation. In all the other scenarios, PFC acts similar to SS encoding.

The first operation, absorption, is used to allow a cell to take over the adjacent cell within a segment. Originally, cell $c_{i,j}$ in segment S_i is assigned to d_j. If the absorption operation is applied to $c_{i,j}$, then the right-adjacent cell $c_{i,j'}$,

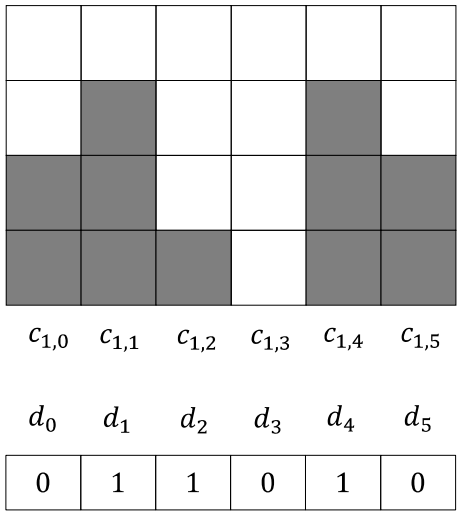

Fig. 1: Mapping a PFC segment to data bit-values.

where $j' = (j+1) \bmod k$, is reassigned from $d_{j'}$ to d_j. We can view the result of the operation as $d_{j'}$ has either been reassigned to a cell in the nearest available segment S_{i+x}(x is some positive integer) or has become unassigned. Both are acceptable if we assume that an unassigned bit has a value of 0. Consider the segment in Fig. 1. Observe that in the next write operation for d_4, the encoding process will increase the cell-value of $c_{1,4}$ which makes the cell full. This will invoke the absorption operation and reassign $c_{1,5}$ to d_4 as shown in Fig. 2.

The absorption operation is incomplete on its own. Although we cannot observe any error in the previous example, this is only because cell $c_{i,5}$, referred to as the *adsorbate*, has an even cell-value. This causes no problem because absorbing a cell with even parity does not affect the value of the bit assigned to the absorber. However, this is not true when the adsorbate has an odd parity. To remedy this conundrum, we invoke the revival operation to preserve data integrity by applying a cell write to "revive" the bit previously assigned to the adsorbate.

We will discuss two approaches for the revival operation. We will follow the same symbols and notation as used in the discussion of the absorption operation. The first is a simple and does not require any change to the absorption operation. This operation simply increases the cell-value of $c_{i,j'}$ and $c_{i+x,j'}$ by 1 each. Take note that cell $c_{i+x,j'}$ of segment S_{i+x} is the cell to which $d_{j'}$ is newly assigned. This approach is the same as the one discussed in [1]. An example can be seen in Fig. 3.

A segment that is one cell write shy from being full has all its cells assigned to one bit. Furthermore, the bit assigned to a full segment has a bit-value of 0, assuming k is even. This implies we can ignore full segments as they have zero significance to the value of the data. The same goes for empty segments. Thus, we will only focus on active

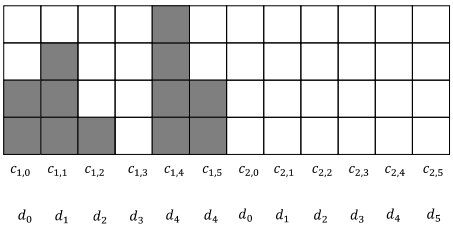

Fig. 2: Updating d_4 causes the (even parity) d_5 to be absorbed.

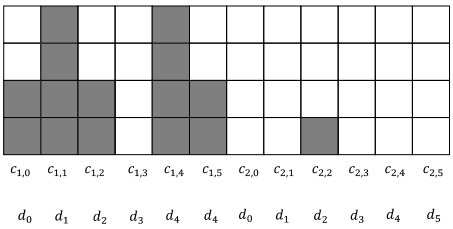

Fig. 3: Applying the first approach of the revival operation to d_2.

segments. A bit can only be assigned in one segment at a time. The core of the decoding function is to determine which cells are absorbed and which are not. This will enable us to know the assingment of cells to bits. We can state that a cell is *independent*, i.e. not absorbed, if its left adjacent cell is not full. Once assignment of cells to bits is known, computing the bit-value is simply done by computing the parity of the appropriate cell-values.

There are three factors to the write deficiency. The first is the unused cells that were not enough to form one segment which is at most $k-1$ cells. The second factor is the extra two cell writes used for each revival operation. In a segment, we can apply at most $k-1$ revival operations. Overall, this contributes to at most $O(n)$ to the write deficiency. The third factor is from the active segments left upon block erasure. There can be at most $k-1$ active segments. A loose upper bound to its contribution to write deficiency is $O(k^2q)$. Hence the total write deficiency is at most $O(n+k^2q+kq) = O(n+k^2q)$.

As stated before, the first approach for the revival operation comes with an overhead cost that is at most $O(n)$, which occurs when the adsorbate cell, i.e., $c_{i,j'}$ as stated in the previous discussion, has an odd parity. The second approach remedy this problem by delaying the cell write to $c_{i,j}$, the absorber cell, that would have rendered it full. Instead, it directly applies a cell write to $c_{i+x,j'}$. In doing so, it transfers the data information of $d_{j'}$ from $c_{i,j'}$ to $c_{i+x,j'}$ and causes $c_{i,j'}$ to be implicitly absorbed by $c_{i,j}$. This saves PFC from applying the extra two cell writes as stated in the first approach. Hence removing the overhead cost. The second approach of the revival operation is illustrated in Fig. 4.

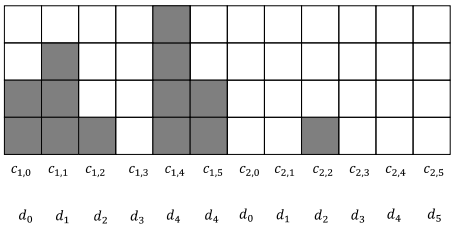

Fig. 4: Applying the second approach of the revival operation to d_2.

The second approach of the revival operation requires modification in determining absorbed and independent cells. Let the set $T = \{S_{a_1}, \ldots, S_{a_r}\}$ be the set of active segments in state of the block C. We first process the rightmost segment S_{a_r} using the same rule to determine between absorbed and independent cells as the one in the first approach. Afterwards, starting from $S_{a_{r-1}}$ to S_{a_1}, we follow a slightly modified rule to determine between absorbed and independent cells. For every active cell $c_{a_i,j}$ in segment S_{a_i} that is labelled as independent by the previous rule, the modified rule dictates that if d_j is already assigned to a segment S_{a_x}, $x > i$, then $c_{a_i,j}$ is labelled as absorbed. Afterwards, we can proceed similarly as the decoding procedure in the first approach and compute of the bit-value of each bit of the data. By removing the overhead cost of using the revival operation, we are able to reduce the upper bound write deficiency of PFC from $O(n+k^2q)$ to $O(k^2q)$. This performance is comparable to the first phase encoding of ILIFC [4].

4. Combining PFC and BIFC

In the previous section, the implementation of the second approach removed one of the factors of write deficiency of PFC. In this section, we will discuss a method to diminish the effect of the first factor, the cells that was never used as a segment, to the write deficiency. In our modified flash code we combine PFC and BIFC [10], and refer to this as the Phoenix Flash Code with BIFC (PFCB). We first divide the block into two major partitions. In the left partition, we fit as many PFC segments as possible, i.e., $m = \lfloor n/k \rfloor$ segments. The (possibly empty) right partition is then allocated for the BIFC slices. See Fig. 5 for this. This implies that BIFC only uses the cells which are not enough to form a segment, thus addressing the write deficiency due to the $n \mod k$ cells in the original Phoenix Flash Code. Note that if there are further remainder cells that cannot form a BIFC slice, these cells are incorporated in the PFC partition.

Whenever it is possible to update some i-th bit using the PFC partition, the appropriate PFC segment(s) is updated. Otherwise an appropriate BIFC slice is updated. Only when both options are not possible does a block erasure occur.

Fig. 5: Visual representation of PFCB and BIFC partitions.

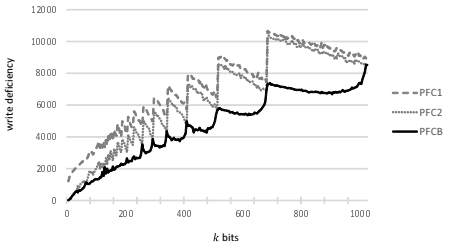

Fig. 6: Simulation results using uniform distribution.

When n is significantly larger than k, the portion for BIFC may seem insignificant in relation to the entire block. However, there are still more than enough cells to form many slices to continue the encoding process than it would have been without the BIFC partition.

5. Results

We simulated the performance of different flash codes and compared the resulting write deficiencies. In the simulations, the block size was fixed at 2048 cells and the maximum charge of a cell was set to 7. The bit size k was varied, and for each k value, 30 experiments were run, and the average write deficiencies were reported.

We tested the performance of the flash codes in two different distributions. In the uniform distribution, each bit has an equal probability of $1/k$ of being updated. We first investigate the improvements brought about by modifying the revival operation of the original PFC to incorporate the second approach and also the improvements brought about by incorporating the BIFC partition. Fig. 6 and 7 clearly illustrates the superiority of PFCB compared to the other coding scheme, even against the previous best DMFC. As a side note, the write deficiency graphs of ILIFC and BIFC each shows a sharp increase at some k value. This occurs when a block has not enough cells to accommodated one slice for each bit of the data. We also test the performance of the PFCB against other flash codes in a non-uniform distribution. Here 1 bit has 50% probability of being updated, while the rest of the other bits have a uniform $0.5/(k-1)$ probability of being updated which is illustrated in Fig. 8.

6. Conclusion

In this study, we propose a new flash code that combines the Phoenix Flash Code (PFC) and the Binary Indexed Flash Code (BIFC). Simulation results indicate that it has a significantly better write deficiency than existing flash codes in literature. Generally, a better write deficiency leads to a longer lifespan for flash memory devices.

It would be interesting to investigate other techniques for lowering the write deficiency of flash codes, by perhaps developing new operations. Future studies can also explore other ways of combining two or more flash codes, and evaluating the resulting performance.

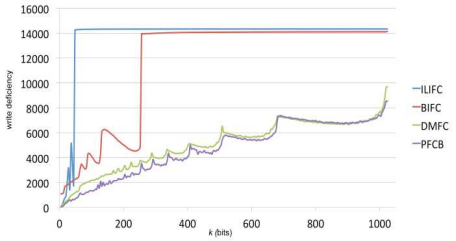

Fig. 7: Simulation results using uniform distribution.

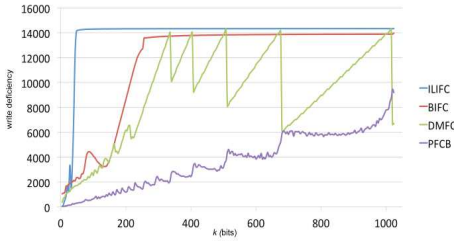

Fig. 8: Simulation results using one bit dominating 50% of the updates.

References

[1] G. Corneby, L. Sanchez, M. J. Tan, P. Fernandez, and Y. Kaji, "Phoenix flash code: introducing the absorption and revival operations for reducing flash memory write deficiency," in *Proc. (NCITE 2013)*, 2013, pp. 209–215.

[2] A. Jiang, V. Bohossian, and J. Bruck, "Rewriting codes for joint information storage in flash memories," *IEEE Transactions on Information Theory*, vol. 56, pp.5300–5313, Oct. 2010.

[3] Y. Kaji, "The expected write deficiency of index-less flash codes and their improvement," *IEICE Trans. on Fundamentals*, vol. E95-A, no. 12, pp. 2130–2138, Dec. 2012.

[4] H. Mahdavifar, P. Siegel, A. Vardy, J. Wolf, and E. Yaakobi, "A nearly optimal construction of flash codes," in *Proc. ISIT'09*, 2009, pp. 1239–1243.

[5] A. Olson, and D. Langlois, "Solid state drives data reliability and lifetime," In Imation Corporation White Pater, 2008.

[6] R. Suzuki, and T. Wadayama. "Layered index-less indexed flash codes for improving average performance," in *Proc. ISIT'11*, 2011, pp. 2138–2142.

[7] M. J. Tan, and Y. Kaji, "Uniform-compartment flash code and binary-indexed flash code," in *IEICE Technical Report*, 2012, pp. 25–30.

[8] M. J. Tan, and Y. Kaji, "Flash code utilizing binary-indexed slice encoding and resizable-clusters," *IEICE Trans. on Fundamentals*, vol. E96-A, no. 12, pp. 2360–2367, Dec. 2013.

[9] M. J. Tan, P. Fernandez, N. Salazar, J. Ty, and Y. Kaji, "Flash code with dual modes of encoding,". in *Pre-Proc. WCTP'13*, 2013.