TAF Journal of Advances in Technology and Engineering Studies JATER
2017, 3(1): 11-18

PRIMARY RESEARCH

Flash codes with multibit update for asymmetrical write memory device

Kenneth Ivanson D. Laurel [1*], **Proceso L. Fernandez** [2]

[1, 2] Ateneo de Manila University, Quezon City, Philippines

Index Terms
Flash
Flash Code
Flash Memory
Multibit Update
Unibit Update

Received: 30 August 2016
Accepted: 25 September 2016
Published: 12 February 2017

***Abstract*—** A flash code is a coding mechanism used to store and retrieve information in a flash memory, which is simply an array of flash cells. Because of the write asymmetry property of flash cells, a flash code has to be designed carefully in order to efficiently make use of the limited number of program-erase cycles that the flash cells can physically tolerate. From the initial studies on unibit update flash codes, more recent researches have introduced the multibit update mechanism for more efficient flash codes. In this paper, we propose three different new multibit update flash codes. These flash codes were simulated in Java and compared against existing multibit update flash codes using the mean data update count as the main metric for evaluation. The results show that the proposed flash codes are very competitive with the existing multibit update flash codes, with the third proposed new flash code having superior performance for some range of data vector lengths. This indicates that the proposed flash codes make very efficient use of the flash memory cells and, thus, may be helpful in extending the lifetime of flash devices.

I. INTRODUCTION

A. Flash Codes with Multibit Update for Asymmetrical Write Memory Devices

Flash memory is a non-volatile type of memory that is used in many gadgets and appliances. It consists of many partitions, called blocks, each of which is an array of n flash cells capable of storing one of several possible electric charges that can be discretized into q levels 0,1,...,q-1. The values in the cells are used to represent some k-bit data. It is typical to abstract flash memory block as a cell state vector C = (c0,c1,...,c_n-1), where each ci ∈ 0,1,...,q-1, and the contained digital data as an information vector V = (v0,v1,...,vk-1), where each vi ∈ $0, 1$. Each block can further be logically partitioned into smaller groups of contiguous cells called slices.

In order to properly store data in and interpret data from a flash memory, a flash code is required. Formally, a flash code is described by a pair of functions F = (D,E) where D is the decoding function and E is the encoding function. The decoding function D(C):0,1,...,q-1^n → 0,1k gives the k-bit information vector V represented by the given current cell state vector C. The encoding function, on the other hand, is used to update the cell state vector C to some new vector C' = (c'0,c'1,...,c'$n-1$) in order to reflect the corresponding new k-bit information vector V' = (v'0,v'1,...,v'k-1) that replaces the previous vector V. This is typically done by adding charges, by electron injection, to one or more flash cells in a process that is abstractly referred to as a cell write.

Two different frameworks for the encoding function have been previously proposed. In the unibit update framework, the function E(i,C):0,1,...,k x 0,1,...,q-1^n → 0,1,...,q-1^n ∪ ϵ requires as input the index i, which determines which single bit vi from V will be updated, and also the current cell state vector C. Under this framework, only one bit from a previous information vector can be updated at every single application of this function. On the other hand, in the multibit update framework, the encoding function E(V',C):0,1^k x 0,1,...,q-1^n → 0,1,...,q-1^n ∪ $\mathcal{E}\epsilon$ requires the target new information vector V' to be specified. This enables multiple bits

*Corresponding author: Kenneth Ivanson D. Laurel
†Email: lerual.nosnavi@gmail.com

of the previous information vector V to be simultaneously updated even with only a single application of the encoding function.

In both encoding frameworks, the output is either a new cell state vector C' ≠ C (with the constraint that c'i ≥ ci for all corresponding elements between the two vectors) or the block erasure ε in case the encoding constraint cannot be satisfied. The constraint is based on the physical write asymmetric property of flash cells, wherein it is easy to add a charge to an individual cell, but removing a charge requires that all the charges in all the cells of the block be emptied. This emptying of the charges resets the cell state vector to the empty state 0^n, and this resetting process is often referred to as a block erasure.

While it is not prohibited to perform a block erasure, each application of such operation slightly degrades the flash cells physically. There is, thus, a limit to the maximum number of allowable block erasures that a flash memory can accommodate before becoming worn out. It is therefore desirable to delay the need to perform a block erasure as long as possible. This, however, requires that the encoding function of the flash code is intelligently designed to have such feature.

This study aims to introduce new multibit update flash codes that may perform competitively with the state of the art, i.e., delay the application of block erasures similar to existing multibit update flash codes. To do this, we conceptualize new ways of encoding and decoding information in a write asymmetric context, implement these concepts, perform computer simulations and then measure the mean data update count. These will be described in greater detail later. We are hoping that by exploring new flash codes that utilize multibit update, we will be able to propose new ideas that can bring about a more efficient utilization of flash memory cells and thus help extend the lifespan of flash memory devices.

II. REVIEW OF RELATED LITERATURE

Flash codes were motivated by the study conducted by [1] in "How to reuse a 'write-once' memory". The purpose of their study is to explore and demonstrate how a memory can be rewritten many times in Write-Once Memory (WOM).

The main aim for flash codes is to code and decode efficiently in order to delay the occurrence of block erasure as much as possible. In this section, several flash codes are presented, including unibit update and multibit update flash codes, with the latter described in greater detail.

Flash code was first studied by [1]. Here, they used flash codes to represent data when k=2 wherein each bit is located at either end of the memory. [2] developed another flash code in 2008 that can be used when n = k ≥ 3 and 3 ≤ k ≤ 6. They introduced Indexed Codes. This can represent data for a general k bits. In 2008, [3] developed a code that can represent data for arbitrary values of k. In that study, they constructed an extension to the earlier work of [2] to represent multiple dimensions.

Many unibit update flash codes have been proposed since then. These include the Indexless Indexed Flash Codes (ILIFC) [4], Layered Indexless Indexed Flash Codes (LILIFC) [5], Binary Indexed Flash Code [6], K-Partition Flash Code (KPFC), the Phoenix Flash Code [7], Layered Indexless Indexed Flash Code with Absorption (LILIFCWA) [8] and a few more [9], [10] and [11].

More recently, multibit update flash codes have been proposed. Under this framework, it is possible to update several bits of the information vector using only one application of the encoding function. The two flash codes that have been introduced are the Sequential Cascade Flash Code (SCFC) and the Circular Pair Flash Code (CPFC).

In the SCFC, which was introduced by Bautista et al., a cell is allocated for each of the k bits of information to be encoded. Encoding and decoding are done by checking the cells of a block sequentially so that logical indexing is not required. Specifically, this flash code decodes by reading the cells from left to right, ignoring full cells. A full cell is a cell with charge q-1, the maximum possible charge. The parity of the charge of each cell for the first k non-full cells is used to determine the value of the data vector. An even parity means that the corresponding data vector bit has a value of 0, while an odd parity implies a value of 1 (Refer to Figure 1).

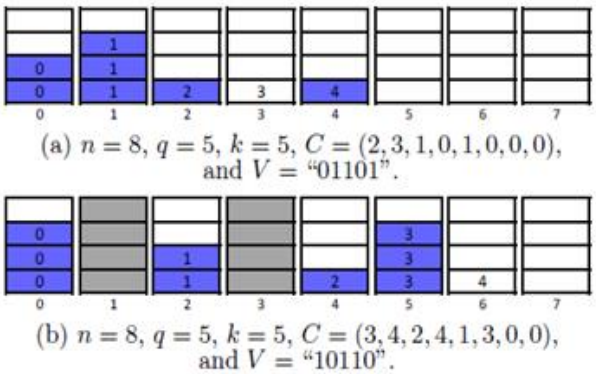


Fig. 1 . SCFC decoding example

ISSN: 2414-4592
DOI: 10.20474/jater-3.1.3

INPUT: $C = (c_0, c_1, \ldots, c_{n-1}), c_j \in \{0, 1, \ldots, q-1\}$
OUTPUT: $V = (v_0, v_1, \ldots, v_{k-1})$

```
1   V = (v_0, v_1, ..., v_{k-1}) ← (0, 0, ..., 0)
2   i ← 0
3   j ← 0
4   while (i < k) and (j < n)
5       if (c_j < q-1)
6           v_i ← parity(c_j)
7           i ← i + 1
8       j ← j + 1
9   return V
```

Fig. 2 . SCFC decoding map

To encode using SCFC, the target k-bit data vector is written to match the parity of the charge of the corresponding first k non-full cells. A cascade occurs if the cell write makes the cell full. Figure 3 illustrates an example of encoding in SCFC. The example shows nine iterations of update until a block erasure. V' is the target data vector for each iteration, t is the number of bits to be updated from the previous V', and ω is the number of cell writes performed for each iteration.

The initial state of the data vector is (0,0,0,0), or simply 0000 for shorthand. On the first iteration, t=3 because three bits were changed from 0000 to 1011. The lighter blue cells are those cells that were encoded during a given iteration. The darker blue cells are cells that were already encoded from the previous iteration/s. Gray cells are full cells.

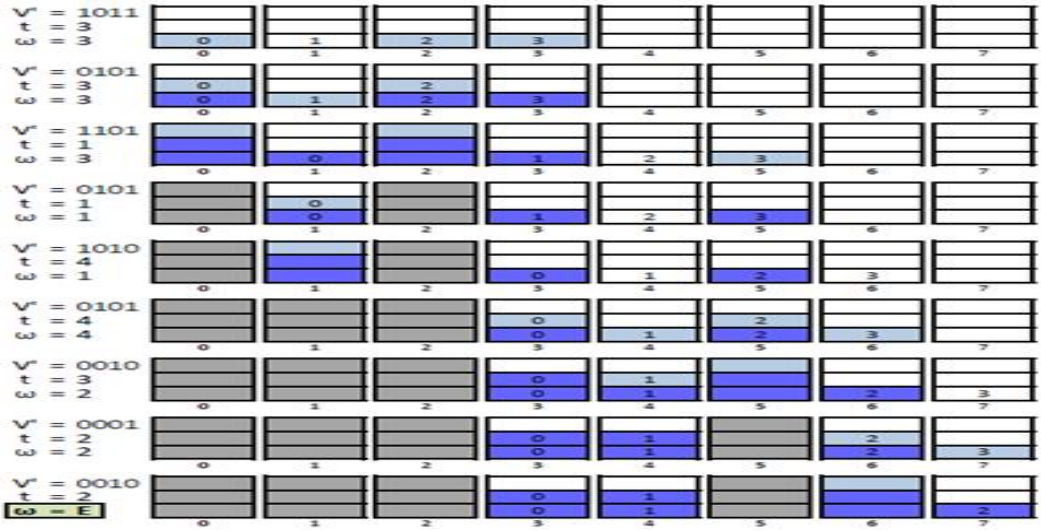


Fig. 3 . SCFC encoding example

INPUT: $V' = (v'_0, v'_1, \ldots, v'_{k-1}), v'_i \in \{0, 1\}$
$C = (c_0, c_1, \ldots, c_{n-1}), c_j \in \{0, 1, \ldots, q-1\}$
OUTPUT: $C' = (c'_0, c'_1, \ldots, c'_{n-1}), c'_j \in \{0, 1, \ldots, q-1\}$
or the block erasure symbol E

```
1   C' = (c'_0, c'_1, ..., c'_{n-1}) ← (c_0, c_1, ..., c_{n-1})
2   i ← 0
3   j ← 0
4   while (i < k) and (j < n)
5       if (c_j < q-1)
6           if (parity(c_j) = v'_i)
7               i ← i + 1
8               j ← j + 1
9           else
10              c'_j ← c_j + 1
11      else
12          j ← j + 1
13  if (i ≠ k)
14      return E
15  else
16      return C'
```

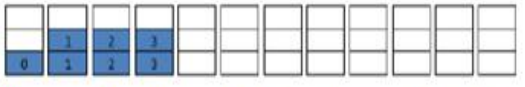
Fig. 4 . SCFC encoding map (Bautista and Fernandez, 2014)

SCFC requires at least k active cells to represent the data vector V of length k. If the number of current active cells does not match the count of bits in the data vector, then a block erasure occurs. Refer to Figures 2 and 4 for the SCFC decoding and encoding maps.

The Circular Pair Flash Code (CPFC) is the second multibit update flash code, and this was introduced by [12]. In this flash code, a block of n cells is divided into 2k slices, wherein a slice can be of type single or pair. A single slice is for unibit update while the pair slice is for pair bits update. This requires a block to be of size n ≥ 2k. Each bit vi of the data vector V can be represented by three slices – the first one is the single slice i, the other two are the bit pairs

ISSN: 2414-4592
DOI: 10.20474/jater-3.1.3

(i,i-1) and (i,i+1). To decode, the parity of the three slices represents the data stored for the bit vi . The indices for the previous and next bit pair slice for bit vi are computed as follows: ((i+1) mod k) +k and ((i-1) mod k) +k, respectively (see Figure 5). To encode, the next bit vi+1 and the previous bit vi-1 of the bit vi are checked in circular fashion and are updated if necessary. If the pair slices are full, the single slice is updated when the adjacent bits do not need an update. A block erasure occurs when all the slices for bit vi are full (see Figure 7). Refer to [12] Circular Pair Flash Code for a complete discussion of the CPFC.



(a) $n = 12,\ q = 3,\ k = 4,\ \alpha = 75,\ C = (1,2,2,2,0,0,0,0,0,0,0,0)$ and $V =$ "1, 0, 0, 0"

(b) $n = 12,\ q = 3,\ k = 4,\ \alpha = 75,\ C = (0,0,0,0,2,0,3,2,1,0,3,1)$ and $V =$ "0, 1, 0, 1"

Fig. 5 . CPFC decoding example [12]

INPUT: $C = (c_0, c_1, \ldots c_{n-1}), c_j \in \{0,1, \ldots q-1\}$
OUTPUT: $V = (v_0, v_1, \ldots v_{k-1}), v_i \in \{0,1\}$

$V = (v_0, v_1, \ldots v_{k-1}) \leftarrow (0,0, \ldots, 0)$
$for\ i \leftarrow 0\ to\ k-1$
    $x \leftarrow i + k$
    $y \leftarrow (i - 1 + k)\%k + k$
    $v_i \leftarrow parity(s_i + s_x + s_y)$
$return\ V$

Fig. 6 . CPFC decoding map [12]

	Update Vector (V')	Data Vector (V)	Block Vector (C)
0		(0,0,0,0)	(0,0,0,0,0,0,0,0,0,0,0,0)
1	(0,1,0,0)	(0,1,0,0)	(0,1,0,0,0,0,0,0,0,0,0,0)
2	(0,1,1,0)	(0,0,1,0)	(0,1,0,0,0,0,1,0,0,0,0,0)
3	(0,0,1,1)	(0,0,0,1)	(0,1,0,0,0,0,1,0,1,0,0,0)
4	(1,1,0,0)	(1,1,0,1)	(0,1,0,0,1,0,1,0,1,0,0,0)
5	(1,0,1,0)	(0,1,1,1)	(1,1,1,0,1,0,1,0,1,0,0,0)
6	(0,1,0,0)	(0,0,1,1)	(1,2,1,0,1,0,1,0,1,0,0,0)
7	(1,1,0,0)	(1,1,1,1)	(1,2,1,0,2,0,1,0,1,0,0,0)
8	(0,1,1,1)	(1,0,0,0)	(1,2,1,1,2,0,2,0,1,0,0,0)
9	(1,1,1,0)	(0,1,1,0)	(1,2,2,1,2,1,2,0,1,0,0,0)
10	(0,0,0,1)	(0,1,1,1)	(1,2,2,2,2,1,2,0,1,0,0,0)
11	(1,0,1,0)		Erase Block

Fig. 7 . CPFC encoding example [12]

III. METHODOLOGY

A. Iterative Conceptualization and Design

Based on the published papers, multibit flash codes are better than unibit in terms of the potential number of bits updated per cell writes, so we explored creating and proposing a flash code under the new multibit update framework. The main idea that we explored for the proposed flash codes is to attempt to have a guaranteed small number of cell writes for every application of the encoding function.

This can be done by assigning each cell or group of cells to correspond to a specific configuration of corresponding bits of the data vector. In the extreme case, it is possible to have at most 2 cell updates for updating any k bit data vector by simply assigning each cell to a distinct configuration of the k length data.

However, this would require n ≥ 2k cells to be available in a block, which would put an unrealistic constraint on the minimum size of a block or on the maximum length of the bit vector that can be encoded.

To strike a balance, the k bit data were divided into pairs so that the required number of cells in a block would be significantly smaller, while at the same time still maintaining some reasonable guarantee on the maximum number of cell writes per encoding.

B. Development and Simulation

The concepts and proposed flash codes were testing corresponding java programs for each and performing simulation experiments. For the simulation, a target data vector which is composed of a series of 0s and 1s was randomly generated based on some parameter p (i.e., for each bit, a 0 is generated with probability p, while a 1 has a probability (1-p). The encoding and decoding methods were then overridden to implement the proposed flash codes. For each k, the program was run 30 times.

After each run, the total number of data updates was recorded. The program was repeated with an increasing value of k starting from 4 up to 2048, in increments of 4. After all of the runs have been completed, the total number of data update was averaged to get the mean data update. These computed means were then recorded in a csv file. Table 1 summarizes the parameter values used in simulating the proposed and also the existing multibit update flash codes.

ISSN: 2414-4592
DOI: 10.20474/jater-3.1.3

C. Benchmarking Metrics

The mean data update metric counts the number of applications of the encoding function before a block erasure occurs. Averaging this provides the main metric used to compare the performances of the existing multibit flash codes with those of the proposed flash codes. To illustrate the mean data update, in Figure 10, the data vector was changed from 0000 to 1010 in Iteration 1. This change in the data vector is one data update. Every iteration is one data update. In this example, the number of data updates is 6. Overall, a higher mean data update is preferred, as this implies a more efficient flash code.

INPUT: $V' = (v'_0, v'_1, \dots v'_{k-1}), v'_i \in \{0,1\}$
$C = (c_0, c_1, \dots c_{n-1}), c_j \in \{0,1, \dots q-1\}$
OUTPUT: $C' = (c'_0, c'_1, \dots c'_{n-1}), c'_j \in \{0,1, \dots q-1\}$
or the block erase symbol E

$S' = (s'_0, s'_1, \dots s'_{k*2-1}) = (s_0, s_1, \dots s_{k*2-1})$
$i \leftarrow 0$
while check(V')
  if $v'_i = 1$
    $v'_i \leftarrow 0$
    $x \leftarrow (i+1)\%k$
    $y \leftarrow (i-1+k)\%k$
    $x_i \leftarrow i+k$
    $y_i \leftarrow y+k$
    if $v'_x = 1$ & *!full*(s_{x_i})
      write(s'_{x_i})
      $v'_x \leftarrow 0$
    else if $v'_y = 1$ & *!full*(s_{y_i})
      write(s'_{y_i})
      $v'_y \leftarrow 0$
    else
      if !full(s_i)
        write(s'_i)
      else if !full(s_{x_i})
        write(s'_{x_i})
        $v'_x \leftarrow 1$
      else if !full(s_{y_i})
        write(s'_{y_i})
        $v'_y \leftarrow 1$
      else
        return E
  $i \leftarrow (i+1)\%k$
return C'

Fig. 8 . CPFC encoding map [12]

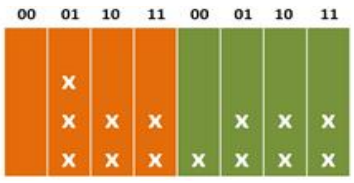


Fig. 9 . TFFC decoding example for 0100

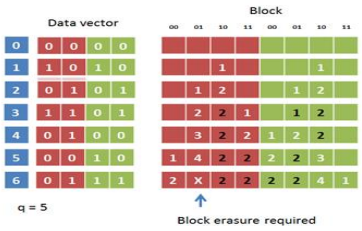


Fig. 10 . TFFC encoding example for a sequence of 6 updates resulting in a block erasure

IV. RESULTS AND ANALYSIS

A. Proposed Flash Codes

Two-Bit Four-Cell Flash Code (TFFC):

In TFFC, a block is partitioned into slices having 4 cells each. The first, second, third and fourth cells in the slice respectively represent the pair bits 00, 01, 10 and 11. The parity of the cell indicates whether or not the pair bit is considered to be active. An active cell has an odd parity charge.

Each of the k/2 pairs of a k-bit data has its own set of slices. To decode, cells within the slices allocated to each k/2 pairs are read sequentially from left to right, ignoring full cells. The parity of the cell is used to determine the pair bit values assigned to it. To encode, the previous active cell with an odd parity charge is first deactivated by adding another charge in order to make the parity even. Then, the cell corresponding to the pair bit to be activated is charged up in order to switch its parity from even to odd. In this flash code, n should at least be twice the k (n ≥ 2k), since every pair of bits has 4 possible configurations (00, 01, 10, and 11).

Decoding example: Figure 9 provides an example of decoding in TFFC. Here, each cell corresponds to a configuration of a bit pair. The specific configurations are shown at the top of the block in the figure. A cell with an odd parity of charge is the active cell. This is the case for the 2nd and 5th cells in Fig. 9. Therefore, the actual data vector is 0100.

Encoding example: An encoding example is provided in Figure 10. In this example, the starting data vector is 0000. Each pair of bits in a 4-bit data vector is represented as a 4-cell slice. The first two bits of the target data vector correspond to the first four cells of the block, while the last two bits of the data vector correspond to the last four cells of the block. To encode 1010 for the first iteration, a charge is added to the 3rd and 7th cells of the block.

B. Two-Bit Three-Cell Flash Code (TTFC)

From TFFC, it was observed that the cell allocated for the first pair bits 00 is unnecessary. It can be implied if the remaining cells all have even parity of charges. Thus three cells are sufficient for every pair of bits. In TTFC, a slice size is always 3 where the first cell represents pair bits 01, second cell represents pair bits 10 and the third cell represents pair bits 11. When there is no odd parity charge cell on a slice, the pair bits 00 is assumed as the active pair bit. Each slice still represents a pair of bits from the k-bit

ISSN: 2414-4592
DOI: 10.20474/jater-3.1.3

information. In this flash code, n should at least be 1.5 times of k (n ≥ 1.5k). Encoding and decoding is similar to TFFC, except that when the new bit data are 00, the active cell will be deactivated and no other cell will be activated.

Decoding example: Figure 11 provides an example of decoding in TTFC. Here, each cell corresponds to a configuration of a bit pair. The specific configurations are shown at the top of the block in the figure. A cell with an odd parity of charge is the active cell. If there are no odd charged cells, the pair bit represented is 00. This is the case for the 1st cell in Figure 11. Therefore, the actual data vector is 0100.

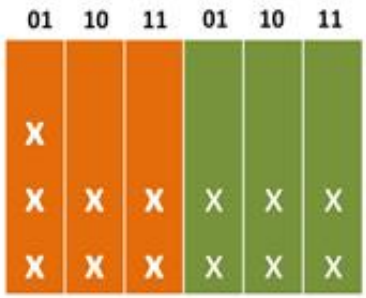


Fig. 11 . TTFC decoding example for 0100

Encoding example: An encoding example is provided in Figure 12. In this example, the starting data vector is 0000. A 2-bit data vector is represented as a 3 cell slice. The first two bits of the target data vector correspond to the first three cells of the block, while the last two bits of the data vector correspond to the last three cells of the block. To encode 0100 for the fourth iteration, a charge is added to the 1st, 3rd and 4th cells of the block. A charge was added to the 1st and 4th cells to activate the cells, while a charge was added to the 3rd cell to deactivate the cell.

Fig. 12 . TTFC encoding example for a sequence of 6 updates resulting in a block erasure

C. Two-Bit Three-Cell First Odd-Parity Flash Code (TFFC-FO)

From TTFC, it was then noticed that the number of cells charged is always two except when 00 is involved, where only one update is needed because of the implied interpretation of 00. A new flash code was introduced where one update may be sufficient to update a data. Similar to TTFC, in TTFC-FO a slice size is also always 3 – the first cell represents pair bits 01, second cell represents pair bits 10 and the third cell represents pair bits 11. When there is no odd charge cell on a slice, it implies that the pair bit currently has the value 00. Each slice still represents a pair of k bits of information. In this flash code, n should at least be 1.5 times of k (i.e., n ≥ 1.5k). Encoding and decoding are done sequentially on each slice, ignoring full cells. In decoding this flash code, the first odd charged cell in a slice represents the data of the slice regardless if the succeeding cells have odd parity of charge.

Fig. 13 . TTFC-FO decoding example for 0100

Fig. 14 . TTFC-FO encoding example for a sequence of 6 updates resulting in a block erasure

Decoding example: Figure 13 provides an example of decoding in TTFC-FO. Here, each cell corresponds to a configuration of a bit pair. The specific configurations are

ISSN: 2414-4592
DOI: 10.20474/jater-3.1.3

shown at the top of the block in the figure. First cell in a slice with an odd parity of charge is the active cell. If there are no odd charged cells, the pair bit represented is 00. This is the case for the 1st cell in Figure 13. Therefore, the actual data vector is 0100.

Encoding example: An encoding example is provided in Figure 14. In this example, the starting data vector is 0000. A 2-bit data vector is represented as a 3-cell slice. The first two bits of the target data vector correspond to the first three cells of the block, while the last two bits of the data vector correspond to the last three cells of the block. To encode 0111 for the last iteration, a charge is added to the 1st, 5th and 6th cells of the block. A charge was added to the 1st and 6th cells to activate the cells while a charge was added to the 5th cell to deactivate the cell.

D. Simulation Results

This section describes the results generated from the computer simulations performed to measure the performance of the new proposed flash codes. Simulations were similarly performed on the other existing flash codes–SCFC and CPFC. At every iteration, the update vector is generated randomly using independent and identically distributed random variables. The number of data update is written in a comma separated values file.

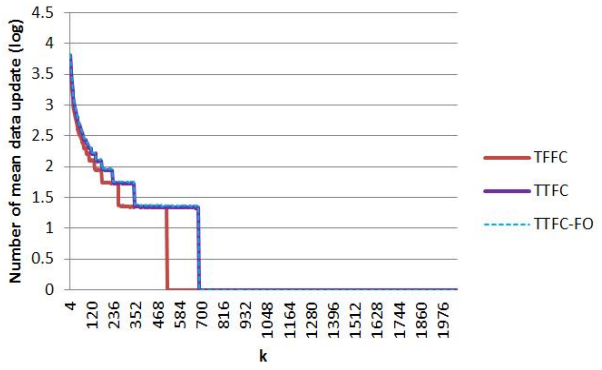


Fig. 15 . Mean data update comparison among the new proposed flash codes

E. Mean Data Update Comparison among the New Proposed Flash Codes

In Figure 15, TTFC (in purple) has a higher mean data update than TFFC (in red) because less cells remained unutilized and only one cell write is needed to change the data to 00, from 01, 10 or 11. TTFC-FO (in blue) has the highest mean data update because in this flash code, one cell write is needed to update from a higher bit pair to lower bit pair combination, e.g., from 11 to 10 or 01, or even from 10 to 01.

F. Mean Data Update Comparison with Previous Multi-bit Flash Codes

Based on Figure 16, TTFC-FO has almost the same mean data update with SCFC alone when k > 684. This is because the cells are not enough for TFFC-FO. In a larger version in Fig. 17, TTFC-FO has a higher mean data update compared to and SCFC up until k=28. In all of the proposed flash codes, the number of cells actually used is determined by the maximum number of cells allocated for each k which can be derived as $(3k/2)^* \lfloor n/(3k/2) \rfloor$ where 3k/2 is the number of slices for the block. This means that, depending on the values of n and k, there will be exactly $n-(3k/2)^* \lfloor n/(3k/2) \rfloor$ cells, which will be unused and unutilized because these cells are not enough to be used for encoding and decoding with the proposed flash codes. These unassigned cells are referred to as remainder cells.

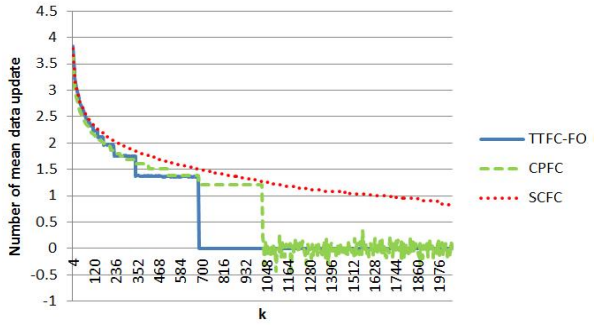


Fig. 16 . Mean data update comparison with previous multibit update flash codes

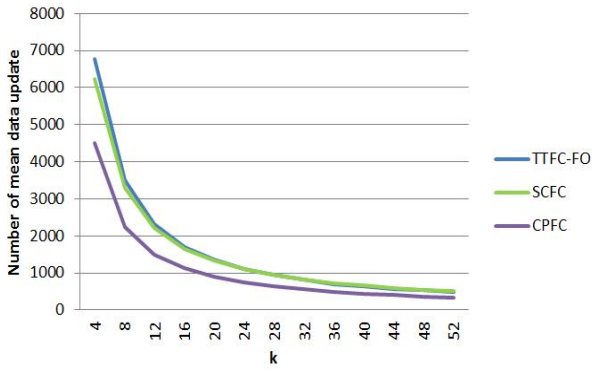


Fig. 17 . Mean data update comparison with previous multibit update flash codes (Zoomed in version)

ISSN: 2414-4592
DOI: 10.20474/jater-3.1.3

V. CONCLUSION

In this paper, three new multibit update flash codes are introduced. The performances of these three new flash codes were measured using computer simulations, and the results were compared against other existing multibit updates such as CPFC and SCFC. The third new proposed flash code (TTFC-FO) proved to be better than the existing multibit update SCFC flash code, in terms of mean data updates, but only for a certain range of values of k for a fixed value of n. For all the new proposed flash codes, there is still a big area for possible improvements because of the currently unutilized remainder cells.

REFERENCES

[1] R. L. Rivest and A. Shamir, “How to reuse a ’write-once’ memory," *Information and Control,* vol. 55, pp. 227-231, 1982.

[2] A. Jiang, V. Bohossian and J. Bruck, “Floating codes for joint information storage in write asymmetric memories,’’ in *IEEE International Symposium on Information Theory,* Barcelona, Spain, 2007. **DOI:** 10.1109/isit.2007.4557381

[3] A. Jiang and J. Bruck, "Joint coding for flash memory storage," in *IEEE International Symposium on Information Theory,* Ontario, Canada, 2008.

[4] E. Yaakobi, A. Vardy, P. Siegel and J. Wolf, "Multidimensional flash codes," in *46th Annual Allerton Conference on Communication, Control, and Computing,* Monticello, IL, 2008. **DOI:** 10.1109/ALLERTON.2008.4797584

[5] H. Mahdavifar, P. Siegel, A. Vardy, J. Wolf and E. Yaakobi, "A nearly optimal construction of flash codes," in *IEEE Inter national Symposium on Information Theory,* Seoul, Korea, 2009. **DOI:** 10.1109/isit.2009.5205973

[6] R. Suzuki and T. Wadayama, "Layered index-less indexed flash codes for improving average performance," in *IEEE International Symposium on Information Theory,* Moscow, Russia, 2011. **DOI:** 10.1109/isit.2011.6033935

[7] M. J. Tan and Y. Kaji, "Uniform compartment flash code and binary-indexed flash code," *IEICE Technical Report, Information Theory,* vol. 112, no. 124, pp. 25-30, 2012.

[8] G. Corneby, L. K. Sanchez, M. J. Tan, Y. Kaji and P. Fernandez, "Phoenix flash code: Introducing the absorption and revival operations for reducing flash memory write deficiency," in *Proceedings of the 11th National Conference for Information Technology Educators,* Dipolog City, Philippines, 2013. **DOI:** 10.1587/transfun.E96.A.2360

[9] A. Maguyon and P. Fernandez, "Introducing sub-block absorption to improve the performance of the layered index less indexed flash code," [Online]. Available: goo.gl/iCOf3h

[10] H. Nagahara and Y. Kaji, "Index-less ash codes with arbitrary small slices," in *International Symposium on Information Theory and Its Applications,* 2012, Hawaii, HI.

[11] H. Esling, R.R. Ortiz and P. Fernandez, "Bi-Modal flash code using index-less indexed flash code and layered index-less indexed flash code," *Advanced Science and Technology Letters,* vol. 35, pp.19-22, 2013. **DOI:** 10.14257/ijmue.2014.9.9.38

[12] J. S. Agustin and P. L. Fernandez, “Circular pair flash code,” [Online]. Available: https://goo.gl/B8DR4L

— This article does not have any appendix. —

ISSN: 2414-4592
DOI: 10.20474/jater-3.1.3