Introducing Sub-Block Absorption to Improve the Performance of the Layered Indexless Indexed Flash Code

Ariel A. Maguyon[1] Proceso L. Fernandez[2]

[1,2] Department of Information Systems and Computer Science, Ateneo de Manila University,
Quezon City, Philippines
{amaguyon, pfernandez}@ateneo.edu

Received 19 June 2014; Revised 5 November 2014; Accepted 24 November 2014

Abstract. Flash codes are used to encode and decode digital information in flash memory. One of the more well-known flash codes in literature is the Layered Indexless Indexed Flash Code (LILIFC). This study explores how to improve the performance of the LILIFC while staying true to its core. A new flash code is proposed, called the Layered Indexless Indexed Flash Code with Absorption (LILIFCWA), which introduces sub-block absorption to the LILIFC encoding function while retaining the original decoding function. Results from computer simulation showed marked improvements gained by LILIFCWA, with lower write deficiency ratios over variable flash memory sub-block sizes. An interesting correlation between the (LILIFCWA versus LILIFC) performance difference and cases involving a dominant bit update was also uncovered. These findings not only prove the superiority of the LILIFCWA over the LILIFC, but may also provide ideas to other researchers on how to improve other existing flash codes.

Keywords: flash code, memory, coding scheme, block erasure

1 Introduction

Flash memory has become an important and ubiquitous storage technology in modern computing [1]. It integrates nicely into the modern paradigm commonly known as the Internet of Things [1], [2]. This type of non-volatile memory has been incorporated into many hardware platforms and portable storage units such as USB flash drives and flash memory cards used in many consumer electronic products. With the popularity of Personal Mobile Devices (PMDs), flash memory has become the preferred secondary storage component over hard disk drives to address limitations in space, speed, and energy requirements of such devices [3].

On many design decisions including those made for PMDs, the choice of flash memory over hard drives is clear. Aside from the small form factor, flash memory has no moving parts, which means less power consumption and virtual immunity to shock – a perennial problem of hard drives. Its access time is also significantly lower (usually in the order of 103 times [3]) compared to hard drives. However, flash memory has its share of disadvantages. For one, its price is still considerably higher (approximately $2/GB) compared to the price of hard drives (approximately $0.09/GB) [3]. Another disadvantage, though less known to most consumers, is its lifespan. This will be explained shortly.

Complementary Metal Oxide Semiconductor (CMOS) transistors make up the core storage units of flash memory [3], [4]. Typically, each transistor has a floating-gate cell that supports two or more charge levels, starting from charge level 0. A cell could represent one or more data bits depending on its maximum charge [4], [5]. A Single-Level Cell (SLC), for instance, can store one bit of data because it only has two charge levels (0 and 1) while a Multi-Level Cell (MLC) can store more, usually up to 4 bits, in commercially-available devices [4], [5], [6]. Cells are further grouped into blocks [4], [5], [7] (also called erase blocks [6]). The number of cells and blocks contained in an actual device is highly-dependent on capacity, manufacturing process, and other parameters.

It is important to note that a cell's charge level can only be increased (i.e., from charge level 0 to 1, or from 1 to 2, etc.), but not decreased (i.e., from 2 to 1). This hardware constraint is often referred to as write asymmetry in flash memory. Also, charge levels cannot be reset back to zero on a per cell basis. To reset one cell's charge level, the entire block to which it belongs must perform an operation known as a block erase [7], [8].

As mentioned, flash memory has a lifespan, and this limitation is also hardware in nature. A device's lifespan is constrained by the number of write-erase cycles for an erase block. That number is about 10,000 for MLC and 100,000 for SLC [3], [4]. Once the cycle threshold is reached, memory cells begin to degrade and the device eventually becomes unusable.

One way to counteract the write-erase cycle limitation is to design efficient schemes used for encoding information to flash memory. Such coding schemes, more popularly known as floating codes [5], [7] or flash codes [6], [7], [8], [9], [10], [11], have recently attracted the attention of researchers, particularly computer scientists, because of the challenges involved in their development [6]. The Indexless Indexed Flash Code (ILIFC) [7] and Layered Indexless Indexed Flash Code (LILIFC) [6] are just two of the popular coding schemes discussed in this and numerous other papers. More advanced schemes including Phoenix Flash Code (PFC) [9], Binary-Indexed Flash Code (BIFC) [10], Bi-Modal Flash Code (BMFC) [8], and Multi-mode Flash Code (MMFC) [11] provide better performance but also require more complex algorithms to implement.

1.1 Research Objectives

The primary objective of this paper is to show how the popular coding scheme known as the Layered Indexless Indexed Flash Code could be pushed further to improve performance. The authors further intended to adhere as close as possible to the original LILIFC, so as to maintain its advantage in simplicity and, in many cases, speed. Hence, more complex coding schemes, such as hybrids used in [8], [9], [10], [11] were not considered.

1.2 Significance of the Study

The authors believe that small but significant improvements to LILIFC and/or ILIFC would be helpful to numerous researchers making use of these popular flash codes. The concept of delaying block erasure, for as long as possible, is one of the main driving principles in this research. It is hoped that this would spawn future studies incorporating the same idea.

1.3 Preliminaries

This subsection discusses the basic concepts as well as the theoretical background used throughout this paper. It is important for the reader to first become familiar with the mathematical abstractions presented here in order to understand the context of subsequent discussions.

As mentioned earlier, a flash code is a mechanism used for encoding and decoding digital information in flash memory. Such memory typically consists of many blocks of flash cells. It is sufficient to consider how a flash code works on a single block since this mechanism is just similarly applied to the remaining blocks.

A block may be modeled as an array of n cells. Each cell can hold some charge, abstractly represented by an integer from $A_q = \{0, 1, \ldots, q\text{-}1\}$. Thus, the state of a block may be represented by some vector of integers $C = (c_0, c_1, \ldots, c_{n-1})$, where $c_i \in A_q$ and hence $C \in A_q^n$. Suppose $D = (d_0, d_1, \ldots, d_{k-1})$, where $c_i \in \{0,1\}$ and $D \in \{0,1\}^k$, is some k-bit information stored in the block. The underlying flash code is the mechanism that allows the information vector to be derived from the given state vector (decoding) and also enables the flash device to properly produce a new state vector whenever the information vector is updated (encoding).

The two functions D and E of the flash code $F(D,E)$ define the decoding and encoding operations respectively. Formally, $D : A_q^n \rightarrow \{0,1\}^k$, guaranteeing that every possible block state can be interpreted unambiguously. On the other hand, the encoding function $E : A_q^n \times \{0, 1, \ldots, k\text{-}1\} \rightarrow A_q^n \cup \{\mathbf{E}\}$ takes as its input the current block state $C = (c_0, c_1, \ldots, c_{n-1})$ and the specific index i of the bit to be updated from the information vector $D = (d_0, d_1, \ldots, d_{k-1})$, to either produce a new block state $C' = (c'_0, c'_1, \ldots, c'_{n-1})$, where $C' \neq C$, or return the block erasure token **E** if this is not possible. The write asymmetry property is expressed in the following constraint: $c'_j \geq c_j \; \forall j \in \{0, 1, \ldots, n\text{-}1\}$.

The performance of a flash code is usually measured using the *write deficiency*, introduced in [7], or its normalized version, the *write deficiency ratio* [8], [9], [11]. The latter is used in this paper, and is computed using the formula:

$$\hat{\delta}(F) = \frac{n(q-1)-t}{n(q-1)} \tag{1}$$

where n is the number of cells in the block, q is the number of different charge levels possible in a cell, and t is the actual number of bit updates accommodated by the flash code before a block erasure occurs. The ideal value of the write deficiency ratio is 0, while the worst possible case has a value of 1.

In this study, the partitioning technique used in both ILIFC and LILIFC is adopted. As such, a block is further partitioned into logical groups called *sub-blocks*. Each sub-block consists of contiguous cells, and a sub-block is either *empty* (all cells have a charge of 0), *full* (all cells have a charge of q-1) or *active* (otherwise). Additional terms and definitions necessary are introduced in the containing sections.

2 Review of Related Literature

2.1 Straightforward Coding

One of the simplest ways to store an information vector of word length k to flash memory with an erase block size of n cells is to divide n by k (assume n is a multiple of k for simplicity) to form m sub-blocks and map each bit to the corresponding cell in a particular sub-block [11]. For example, suppose the information vector is composed of four bits (bit 0 to 3) and some flash memory is characterized by $n = 8$ and $q = 2$. In this scenario, $m = 2$. Bit i is mapped to cell number i mod k (i.e., bit 1 is mapped to cell numbers 1 and 5). This scheme is simple and straightforward to implement, but incurs a relatively high write deficiency, especially with a large value of k combined with frequent updates on a few dominant bits.

2.2 Indexed Schemes

A more elaborate flash code is discussed by Jiang, et al. in [5]. In their proposed scheme, the information vector of size k bits is divided into a (normally ≤ 6) groups, $g_0, g_1, \ldots, g_{a-1}$. On the other hand, the erase block of size n is divided into two parts. The first contains *index cells* that indicate which group in the information vector is being referred to. The second part stores the actual data and is further subdivided into b groups, $h_0, h_1, \ldots, h_{b-1}$, such that $b \geq a$. Typically, $\log_q a$ index cells are allocated for each group in the data cells. Initially, group g_i in the information vector maps to group h_i in the erase block. After a number of rewrites and h_i is no longer able to accommodate another change in the group g_i, the next free available group, h_j, in the block will be allocated.

Indexed flash codes are useful. However, the main problem associated with these codes emerges when $n >> k$, in which the write deficiency could also become arbitrarily large [7].

2.3 Indexless Schemes

ILIFC. The popular scheme known as Indexless Indexed Flash Code, proposed by Mahdavifar et al. [7], implemented a novel approach to encoding the information vector in an erase block without incurring an additional overhead needed to store the indices [12]. ILIFC accomplished this using the following technique. The erase block is first logically subdivided into m sub-blocks, computed as $m = n/k$, where k is the word length of the information vector. Implicitly, k also determines the size of each sub-block. The charge level of each cell in the sub-block is represented by the k-tuple cell state vector $S = (s_0, s_1, \ldots, s_{k-1})$, which is just $(c_{jk}, c_{jk+1}, \ldots, c_{jk+k-1})$ for the j^{th} sub-block, and is initially equal to $(0, 0, \ldots, 0)$.

ILIFC encodes the information vector as follows. One sub-block is used to represent one bit of data. A change in the i^{th} bit of the information vector is encoded into a sub-block by increasing the charge level starting from the i^{th} cell of the block and progressing to the next cell i+1 as the previous cell reaches the maximum charge of q-1. This cell programming pattern treats cell numbers in a block as a circular array, wrapping around towards $i = 0$ from $i = k$-1 until the block becomes full (all cells already reached the maximum charge level of q-1) or a block erasure needs to be carried out.

The bit index encoded in a sub-block is decoded by searching for the cell with the highest charge level q-1. The position, i, of that cell within the sub-block yields the index. If there is more than one cell with the maximum charge, the position of the leftmost cell (with charge q-1) provides the index. The use of the term *leftmost* in the preceding sentence should be taken in the context of the index wraparound protocol effective within each sub-block, such that $s_i = q$-1 and $s_{[(i+k-1) \bmod k]}$ is the minimum cell level in the sub-block S.

The bit value stored in a sub-block is determined by the parity of the sum of all charge levels within the sub-block. Mathematically, that would be $(s_0 + s_1 + \ldots + s_{k-1}) \bmod 2$, yielding either a 0 or a 1.

LILIFC. The Layered Indexless Indexed Flash Code discussed exhaustively in [6], enhanced the original ILIFC by implementing a programming pattern that grows horizontally instead of vertically to the right (with wraparound). This scheme makes use of *layers* that serve as boundary for the maximum charge of all cells in a sub-block.

A major advantage of this flash code over ILIFC is the ability to reallocate a sub-block encoded with some index i_y to a different index i_z, under certain circumstances. It must be noted that ILIFC does not have this kind of flexibility. This advantage translates to lower write deficiencies for a given range of k values and better overall performance compared to ILIFC.

Details of the LILIFC follow. To simplify the discussion, however, it is further assumed that n is a multiple of k and that k is an even number. A write to an empty sub-block encoding the index i is made by increasing the charge level of the i^{th} cell of that sub-block by one. The second write to the same sub-block increases the charge of cell i+1, the third write increases the charge of cell i+2, and so on. This programming pattern fills up a sub-block on a layer-by-layer basis. Hence, the layer l of a sub-block is determined by the maximum charge level of all cells in that sub-block. Alternatively, it can be said that a sub-block progresses from layer 0 to q-1. The basic difference in the ILIFC and LILIFC coding schemes is illustrated in Fig. 1. LILIFC sub-blocks are decoded in a slightly different manner compared to ILIFC. The index is determined by looking for the position, i, within the sub-block of the leftmost cell belonging to the sub-block's current layer l. Again, the term *leftmost* is used in a wraparound context and implies that $x_i = l$ and $x_{[(i+k-1) \bmod k]} = l$-1. LILIFC uses the same rule followed by ILIFC to decode a sub-block's bit value.

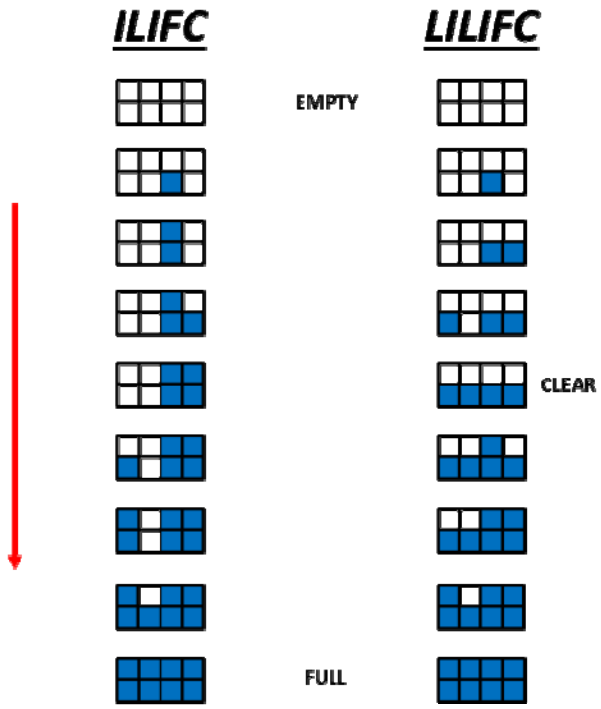


Fig. 1. ILIFC vs. LILIFC cell programming for bit index 2.

An active sub-block is further described to be in the *clear* state if the charge level in all cells is the same, at some fixed layer $l \in \{0, 1, \ldots, q\text{-}2\}$. This characteristic provides flexibility to LILIFC. In the situation when ILIFC can no longer find an empty sub-block and needs a block erase, LILIFC will look for a clear sub-block and encode the new index into it. The old index encoded in that clear sub-block can simply be "forgotten" (a sub-block in a clear state has an even parity, assuming k is even).

3 Methodology

3.1 Enhancing the Original LILIFC Scheme

Modifications made to the LILIFC are detailed in the following subsections. These are explained by a discussion on the basic concept of the block erasure point and how to delay it, as used in the proposed flash code.

Block Erasure Point. The point of block erasure in the original LILIFC coding scheme happens when a bit update is required but there is no sub-block assigned or may be assigned to that bit. The latter occurs when there is no sub-block that is either empty or is in the clear state. This proved to be very limiting for the LILIFC, especially for a large value of the information vector word length, *k*.

Delaying Block Erasure. The authors of this paper wanted to improve the performance of the LILIFC by delaying the block erasure point. As mentioned in the *Research Objectives* section, considered enhancements were only limited to those that did not encode multiple information vector bits in the same sub-block.

Although the block decoding function remained the same as LILIFC's, the encoding was tweaked to work a little differently. Instead of doing a block erase during LILIFC's original block erasure point, the proposed enhancements search for an active, but even-parity sub-block that could be absorbed. The pseudo-code for this scheme is shown in Fig. 2.

Encoding Map

$y = (y_0 \mid y_1 \mid \cdots \mid y_{m-1}) = (x_0 \mid x_1 \mid \cdots \mid x_{m-1})$;

```
for ( j = 0..m-1 ) {
  if ( (¬ clear(xj)) ∧ (index(xj)==i) ) {
    write(yi);
    return(y);
  }
}
for ( l = 0..q-2 ) {
  for ( j = 0..m-1 ) {
    if ( clear(xj) ∧ layer(xj)==l ) {
      write_new(i,yj);
      return(y);
    }
  }
}
for ( j = 0..m-1 ) {
  if ( parity(xj) == 0 ) {
    absorb(i,yj);
    return(y);
  }
}
return E;
```

Fig. 2. The main algorithm used by LILIFWCA to encode a bit into a sub-block.

A major portion of the above pseudo-code was largely based on the LILIFC code mentioned in [6]. The vector $(x_0 \mid x_1 \mid \cdots \mid x_{m-1})$ represents the current state of the erase block while $(y_0 \mid y_1 \mid \cdots \mid y_{m-1})$ represents the changed state. The variable *m* indicates the total number of sub-blocks, while *q* denotes the number of different state levels possible for a cell.

The first `for` loop of the code checks if the index *i* is currently assigned a sub-block. If it is, the `write()` function increments the level in the appropriate cell of that sub-block and returns the revised block. If bit *i* does not have an assigned sub-block, the algorithm looks for a sub-block with a clear state and has the lowest layer level. The function `clear()` checks if a sub-block has a clear state and returns either true or false. On the other hand, `layer()` returns the layer level (highest cell value) in the current sub-block. If such a sub-block is found, the `write_new()` function encodes the new index to that sub-block and assigns it to *i*. At this point, if no clear state sub-blocks are available, LILIFC issues a block erase.

LILIFCWA delays the block erase by looking for active sub-blocks with an even parity in the last for loop of the pseudo-code. The function `parity()` returns the parity value of a given sub-block x_j. If more than one such sub-block is found, LILIFCWA absorbs the sub-block using one of two techniques used in three versions (*please refer to the next subsection for details*) of the proposed flash code. The pseudo-code presented above uses `absorb()` to claim the sub-block it encounters satisfying the criteria on a first-come-first-served basis. Version 3 of LILIFCWA absorbs the sub-block that requires the least number of writes. Finally, LILIFCWA issues a block erase if no active, even-parity sub-block is found.

To further clarify how LILIFCWA works, a sample scenario is provided in Fig. 3.

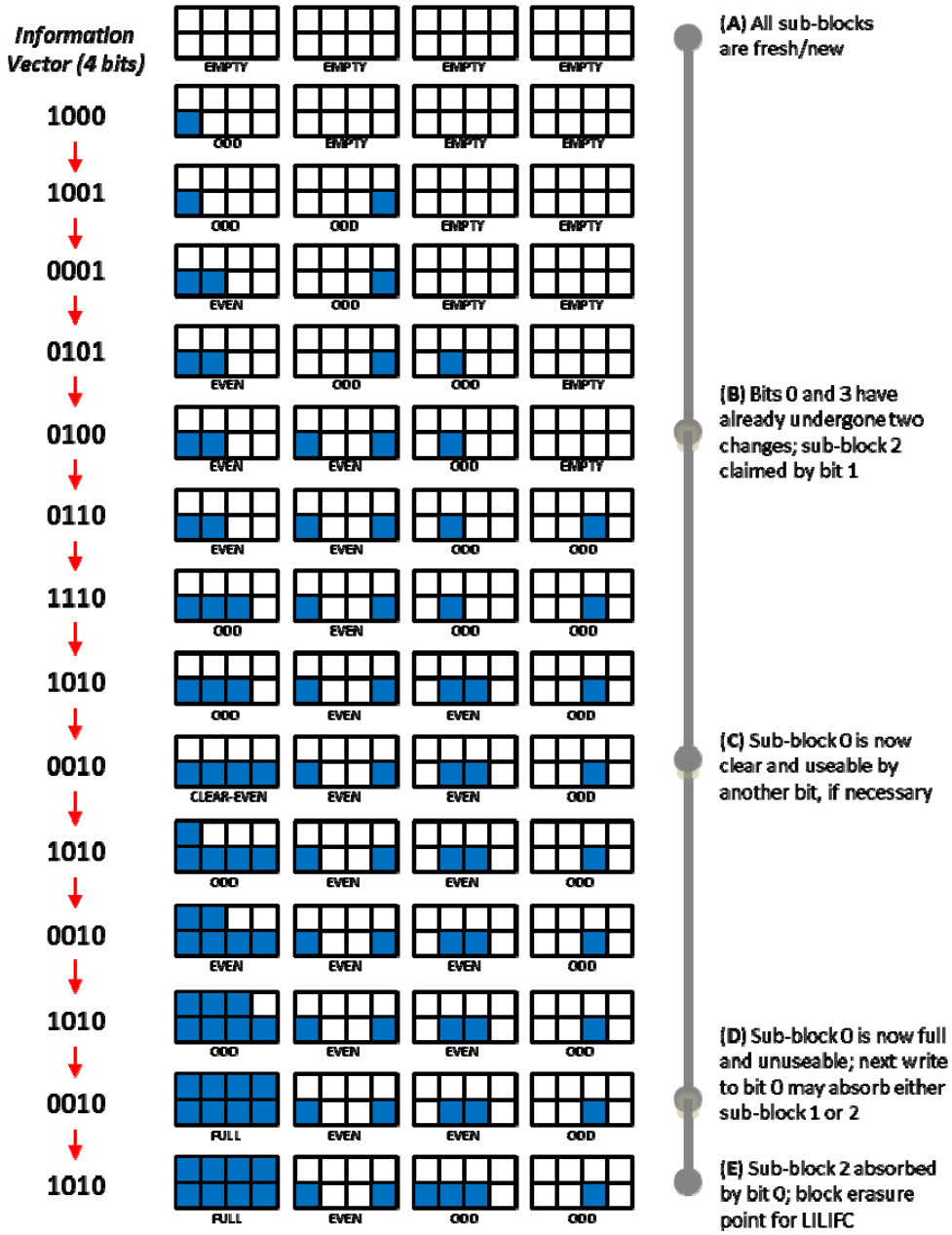


Fig. 3. A sample scenario showing how LILIFCWA works through a series of changes by a 4-bit information vector.

3.2 Measuring Performance via Simulation

Computer simulation, written in the Java programming language, was used to gauge the performance of the proposed flash code against the LILIFC and ILIFIC. Various write pattern scenarios were explored using the program parameters summarized in Table 1.

Table 1. List of relevant program parameters used in the simulations

Description	Variable	Value(s)
Block size (fixed)	n	2048
Charge states (fixed)	q	8
Sub-block size (variable)	k	4, 8, 12, …, 1024
Update probability of dominant bit (variable)		30%, 50%, and 70%

Two basic scenarios were investigated. One involved a steady uniform distribution, where all bits had the same probability 1/*k* of being flipped. The other had a dominant bit, the concept of which, including various degrees of domination, is explained as follows.

- *Steady dominated distribution (30%)* – one bit had a 30% chance of being flipped; the rest had equal chances at (1.0-0.3)/(*k*-1) probability
- *Steady dominated distribution (50%)* – one bit had a 50% chance of being flipped; the rest had equal chances at (1.0-0.5)/(*k*-1) probability
- *Steady dominated distribution (70%)* – one bit had a 70% chance of being flipped; the rest had equal chances at (1.0-0.7)/(*k*-1) probability

The encoding mechanism (at the block erasure point) of the proposed flash code, LILIFCWA, was written in three versions (refer to Fig. 4)

- Version 1.0 – the first available even-parity sub-block is selected, and then the absorption of that sub-block is done by simply clearing the sub-block (i.e., making the charges of all cells in a sub-block equal to the maximum cell charge in that sub-block) and then encoding the new bit in the next layer
- Version 2.0 – the first available even-parity sub-block is selected, and then the absorption of that sub-block is done by making the fewest writes possible to convert the sub-block to the new index; this does not necessarily require clearing the sub-block first
- Version 3.0 – absorption of an even-parity sub-block is done by making the fewest writes possible to convert the sub-block to the new index; the even-parity sub-block selected is the one that requires the least number of writes for such a conversion

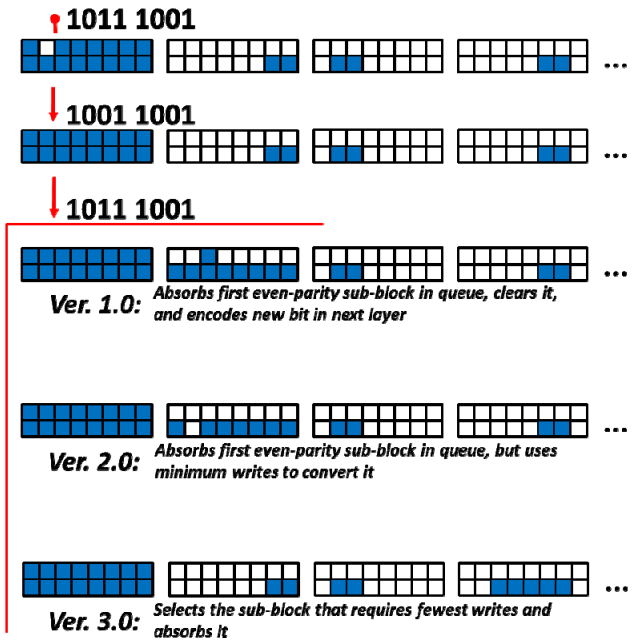


Fig. 4. An illustration of how the three different LILIFCWA versions work with an 8-bit information vector. Due to space constraints, only the first four sub-blocks are shown and it is assumed that the last four are of odd parity, as indicated in the information vector. It is also assumed that the second sub-block from the left is at the head of the even-parity queue.

With each refinement, LILIFCWA was compared to both ILIFC and LILIFC.

4 Results and Analysis

The following subsections examine the performance of LILIFCWA. The discussion used the write deficiency ratio as the central metric to compare the proposed flash code to LILIFC and ILIFC.

4.1 LILIFCWA Write Deficiency Ratio Boundary

The write deficiency ratio of LILIFCWA is always lower than or equal to that of LILIFC's. Mathematically, this relationship can be expressed as

$$\delta_{k,\text{LILIFCWA}} \leq \delta_{k,\text{LILIFC}}, \forall k > 0 \quad (2)$$

This is because the LILIFCWA coding scheme is the same as that of LILIFC's, except at the latter's block erasure point. These results are shown in Table 2 for the case of steady uniform distribution.

Table 2. A comparison of write deficiency ratios for LILIFCWA (Version 3), LIILIFC, and ILIFC sampled over a range of k values for the case of steady uniform distribution

k	$\hat{\delta}_{k,\text{ILIFC}}$	$\hat{\delta}_{k,\text{LILIFC}}$	$\hat{\delta}_{k,\text{LILIFCWA3}}$
4	0.00298	0.00030	0.00030
8	0.01329	0.00170	0.00170
12	0.03484	0.00820	0.00820
16	0.05064	0.00777	0.00777
20	0.06617	0.01679	0.01679
24	0.14921	0.02234	0.02234
28	0.22110	0.02770	0.02770
32	0.10012	0.03316	0.03316
36	0.35625	0.06219	0.06107
40	0.24431	0.07004	0.06519
44	0.13187	0.09712	0.08715
48	0.99292	0.99292	0.29879
52	0.99467	0.99467	0.56061
56	0.99598	0.99598	0.95070
60	0.99658	0.99658	0.98696
64	0.99696	0.99696	0.99273
68	0.99731	0.99731	0.99548
72	0.99749	0.99749	0.99605
76	0.99782	0.99782	0.99698
80	0.99791	0.99791	0.99713

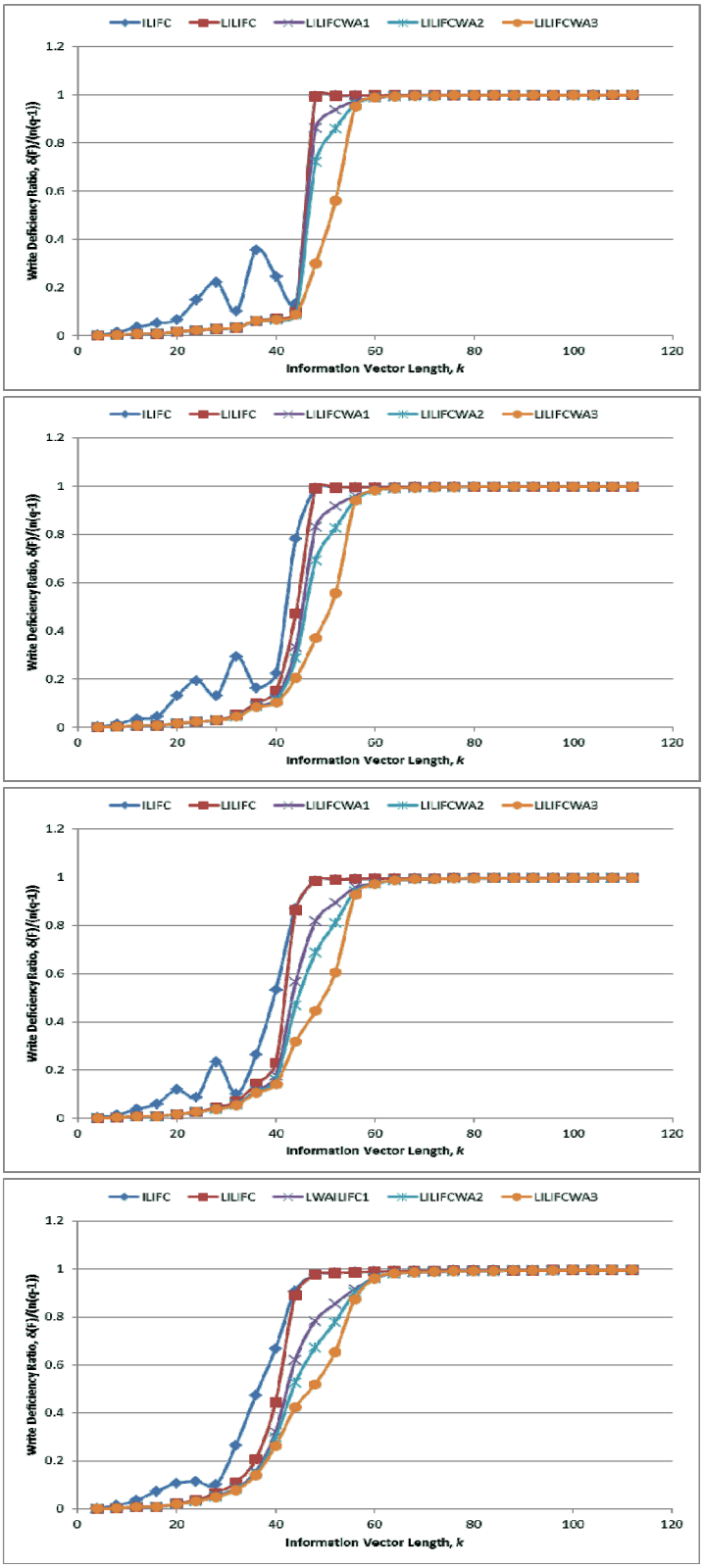
This behavior was largely expected from the proposed flash code because its improvements were based mainly on delaying the block erasure point. That is, there would be no cases where LILIFCWA has a block erasure point that occurs before LILIFC's, given the same sequence of bit updates experienced by the information vector of length k, stored in some block b with a total of n cells.

Although Table 2 only showed the case for the steady uniform distribution, similar behavior was also observed in the dominated (30%, 50%, and 70%) cases.

The write deficiency ratios of ILIFC, LILIFC, and all three versions of LILIFCWA were graphed for comparison and are shown in Fig. 5. From the graph, it is clear that for all (uniform distribution and dominated) cases, LILIFCWA has the lowest write deficiency ratio among the different flash codes considered. It is also evident that version 3 of the proposed flash code (indicated by LILIFCWA3 in Fig. 5) provides the highest performance. Behavioral characteristics of LILIFCWA's performance can also be inferred by examining the graphs in order from (a) to (d). These insights are discussed in the next subsection.

4.2 Difference in Write Deficiency Ratios

For previously proposed flash codes that divide the total number of cells n in a block into sub-blocks of length k, performance is largely dependent on the limitation that $k \leq n^2$ or, more intuitively, $k \leq \sqrt{n}$, assuming equal probability of each bit in the information vector being changed. This boundary for the value of k is referred to as the *degradation point* [11]. For cases where $k > \sqrt{n}$, the graph of the write deficiency ratio becomes asymptotic to 1.0.



(a)

(b)

(c)

(d)

Fig. 5. Results of the computer simulations. (a) Steady Uniform Distribution, (b) Steady Dominated Distribution (30%), (c) Steady Dominated Distribution (50%), and (d) Steady Dominated Distribution (70%). LILIFCWA v.3 provided the highest performance in all four graphs.

For both ILIFC and LILIFC, the degradation point for parameters used in the computer simulations was $k \approx 45$ (since n=2048). LILIFCWA was able to extend that degradation point by approximately 50% ($k \approx 68$). This improvement could be explained by taking a closer look at the block erasure point of LILIFC. Given a steady uniform distribution, there is a 50% probability that an active, even-parity sub-block would be available for absorption, thereby delaying the block erasure point and consequently reducing the write deficiency ratio.

Visual inspection of the resulting graphs provided some additional and interesting insights into the behavior of LILIFCWA. The researchers noticed that with increasing percentage of domination, the difference between the write deficiency ratios of LILIFCWA and LILIFC generally became more prominent, especially in the k range of values between 24 and 72. To verify this observation, the sum of the actual differences between the write deficiency ratios was computed over some range of k values as expressed by the following formula.

$$\sum_{k=24}^{72} \left(\delta_{k,\text{LILIFCWA}} - \delta_{k,\text{LILIFC}} \right) \tag{3}$$

Additional dominated scenarios for both LILIFCWA and LILIFC were included in the simulations to verify the apparent trend in the write deficiency ratios of the two flash codes. More specifically, supporting data were generated for probabilities of 60%, 65%, 80%, 90%, and 95% dominated. The new set of results was consolidated with the original and summarized in Table 3. For each percentage of domination, the difference between the write deficiency ratios of LILIFCWA3 and LILIFC was computed using k = 24, 28, …, 72. The sum of those differences (over the same k range) for each dominated scenario is reflected in the column *Sum of Differences in Write Deficiency Ratio*. The last column, *Average Difference in Write Deficiency Ratio*, was derived by dividing each sum of differences over the total number of k samples used (13, in this case). A visual representation of this trend is provided in Fig. 6.

Table 3. Differences in write deficiency ratios between LILIFCWA (Version 3) and LILIFC with respect to the percentage of domination

Domination (%)	Sum of Differences in Write Deficiency Ratio	Average Difference in Write Deficiency Ratio
Uniform distribution	1.20652	0.09281
30%	1.48178	0.11398
50%	1.71994	0.13230
60%	1.67873	0.12913
65%	1.69893	0.13069
70%	1.71788	0.13214
80%	1.90789	0.14676
90%	2.07328	0.15948
95%	2.16216	0.16632

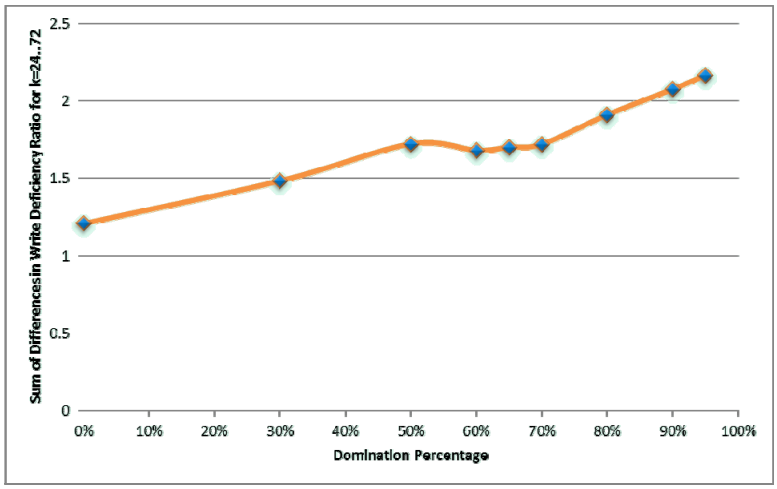


Fig. 6. Graph of differences in write deficiency ratios of LILIFCWA (Version 3) and LILIFC given the percentage of domination.

Given the additional dominated scenarios, the upward trend in the difference looks evident. Again, the explanation for this behavior hinges on the ability of LILIFCWA to absorb sub-blocks. Whereas LILIFC would have already issued a block erase, LILIFCWA is more likely to find and absorb a useable (even-parity, non-flat) sub-block and reuse it many times to encode the dominant bit. Intuitively, the more dominant a bit is, the higher the chance that a useable sub-block absorbed by that bit would be less full (because of much fewer changes made by the previously-encoded, non-dominant bit) before absorption. Each write to the sub-block after absorption would be much more frequent and efficient, helping decrease the write deficiency ratio for LILIFCWA given higher percentages of domination.

5 Conclusion

In this paper a new flash code, called *Layered Indexless Indexed Flash* Code *with Absorption* (LILIFCWA), is proposed. Theoretically, the proposed flash code is superior to the LILIFC and ILIFC. This was validated through computer simulations, which showed significant improvements in the performance of the LILIFCWA compared to LILIFC, while maintaining the latter's core and simplicity. Three versions of LILIFCWA were implemented, and each of these performed better than both ILIFC and LILIFC for cases involving steady uniform distribution and some steady dominated distributions. An interesting result in this study showed an increasing trend in the discrepancy between the write deficiency ratios of LILIFCWA and LILIFC as the percentage of domination rises in the dominated cases. Further studies can explore other possible encoding mechanisms that can be applied at the block erasure point of the ILIFC. Similarly, other flash codes can be considered, and modifications to the encoding mechanisms at their block erasure points can also be investigated.

References

[1] Y.R. Chen and Y.-S. Chen, "Context-Oriented Data Acquisition and Integration Platform for Internet of Things," *Journal of Computers,* vol. 23, no. 4, pp. 1-11, 2012.

[2] P.C. Tseng, "How Ubiquitous Can We Get?," *Journal of Computers,* vol. 24, no. 3, pp. 66-72, October 2013.

[3] J. L. Hennessy and D. A. Patterson, Computer Architecture: A Quantitative Approach, 5th ed., Morgan Kaufmann, 2012.

[4] L. M. Grupp, A. M. Caulfield, J. Coburn, S. Swanson, E. Yaakobi, P. H. Siegel, J. K. Wolf, "Characterizing Flash Memory: Anomalies, Observations, and Applications," in *MICRO*, New York City, New York, USA, 2009.

[5] A. Jiang, V. Bohossian and J. Bruck, "Floating Codes for Joint Information Storage in Write Asymmetric Memories," in *IEEE International Symposium on Information Theory (ISIT)*, Nice, France, 2007.

[6] R. Suzuki and T. Wadayama, "Layered Index-less Indexed Flash Codes for Improving Average Performance," in *IEEE International Symposium on Information Theory (ISIT)*, 2011.

[7] H. Mahdavifar, P. H. Siegel, A. Vardy, J. K. Wolf, E. Yaakobi, "A Nearly Optimal Construction of Flash Codes," in *IEEE International Symposium on Information Theory (ISIT)*, Seoul, Korea, 2009.

[8] H. R. Esling, R. R. L. Ortiz, P. L. Fernandez, "Bi-Modal Flash Code using Index-less Indexed Flash Code and Layered Index-less Indexed Flash Code," in *Advanced Science and Technology Letters*, 2013.

[9] G. N. Corneby, L. K. Sanchez, M. J. Tan, P. Fernandez, Y. Kaji, "Phoenix Flash Code: Introducing the Absorption and Revival Operations for Reducing Flash Memory Write Deficiency," in *11th National Conference on IT Education (NCITE 2013)*, 2013.

[10] M. J. T. Tan and Y. Kaji, "Applying Resizable Cluster Method in Binary-Indexed Flash Code," in *SITA*, 2012.

[11] M. J. Tan, P. Fernandez, N. Salazar, J. Ty, Y. Kaji, "Multi-mode Encoding with Binary-Indexed Flash Code," *IEICE Technical Report IT2012-56,* vol. 112, pp. 41-46, 2013.

[12] Y. Kaji, "The Expected Write Deficiency of Index-Less Indexed Flash Codes," *IEICE Transactions,* Vols. 95-A, no. 12, pp. 2130-2138, 2012.

[13] T. Coughlin, "Flash Memory Enables Faster Applications and Content Access," 22 January 2014. [Online]. Available: http://www.forbes.com/sites/tomcoughlin/2014/01/22/flash-memory-enables-faster-applications-and-content-access/. [Accessed 24 July 2014].