Energy-aware Error Control Coding for Flash Memories

Veera Papirla
Arizona State University, Tempe
AZ 85287, USA
vpapirla@asu.edu

Chaitali Chakrabarti
Arizona State University, Tempe
AZ 85287, USA
chaitali@asu.edu

ABSTRACT

The use of Flash memories in portable embedded systems is ever increasing. This is because of the multi-level storage capability that makes them excellent candidates for high density memory devices. However, cost of writing or programming Flash memories is an order of magnitude higher than traditional memories. In this paper, we design an algorithm to reduce both average write energy and latency in Flash memories. We achieve this by reducing the number of expensive '01' and '10' bit-patterns during error control coding. We show that the algorithm does not change the error correction capability and moreover improves endurance. Simulations results on representative bit-stream traces show that the use of the proposed algorithm saves, on average, 33% of write energy and 31% of latency of Intel MLC NOR Flash memory, and improves the endurance by 24%.

Categories and Subject Descriptors

B.3.2 [**Memory Structures**]: Mass storage; C.4 [**Performance of Systems**]: Fault tolerance

General Terms

Algorithms, Design, Performance

Keywords

Flash memories, Error Control Coding, Low-power design, Endurance

1. INTRODUCTION

Flash memory devices are the most popular form of non-volatile storage in embedded systems today. They are widely used for multimedia data storage such as memory sticks in digital cameras, in MP3 players, etc. As the multi-level storage per Flash memory cell continues to improve, it is even being considered as a general purpose memory solution such as disk caches [1].

DAC '09, July 26-31, 2009, San Francisco, California, USA

Table 1: Delay and Energy measurements for Intel MLC NOR 28F256L18 Flash memory [2].

Write bit pattern	Time	Energy
00	110.00 μs	4.738 μJ
01	644.23 μs	29.531 μJ
10	684.57 μs	31.194 μJ
11	24.93 μs	0.752 μJ
word read	0.02 μs	0.001 μJ

In many modern portable embedded systems, memory devices consume significant portion of the total energy. Reducing the energy consumption of memory is therefore crucial to extending the operation lifetime of portable embedded systems. While the read energy and latency of Flash memories are same as traditional memories [2], the write energy and latency are an order of magnitude higher. In fact, the write energy and latency depend on the data values that have to be written. Table 1 gives the write latency and energy involved for all four bit patterns for one of the most popular 4-level NOR Flash memories from Intel [2]. Similar trends are present in Samsung Flash memories as well. Table 1 clearly shows the large difference in writing '01' & '10' bit-patterns compared to '00' & '11' bit-patterns. This feature was exploited in [2] along with prefix-coding to reduce write energy and latency for storing multimedia files.

Almost all Flash memories have Error Control Coding (ECC) to retrieve the correct information bits in case of error. ECC is used to generate redundant bits which along with the information bits form the codeword to be written into memory [3]. In multi-level Flash memories, the probability of error increases because of closer voltage levels assigned to consecutive logic states. The most widely used ECC in Flash memories are Hamming codes and Reed-Solomon codes [4], [5].

In this paper, we propose a method that builds on top of the ECC in Flash memories to reduce the number of '01' & '10' bit-patterns that are stored in memory. We do this by modifying the codeword if the number of '01' & '10' bit-patterns is large. We formally show that the proposed modification does not change the error correction capability of the ECC. Further more, the proposed algorithm can be used on top of any ECC. We show that this method results in significant reduction in the energy and latency of 'write' operations. Simulation results on representative bit-stream traces for a 4-level Intel MLC NOR Flash memory, show that this method, on average, reduces the energy by 33% and latency by 31%. Finally, the proposed method increases the

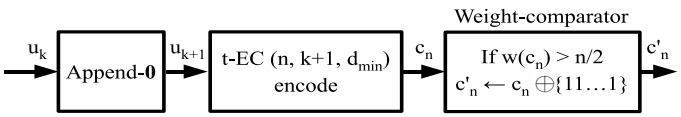

Figure 1: Block diagram of encoder for *Weight-reduction algorithm.*

endurance or lifetime of the memory by 24%. The overhead of this method is fairly small. For NOR Flash memory with block size 512 bits, the overhead is mostly contributed by a Digital Majority Voter (DMV) which cost 340ps in latency and 2.7pJ in energy. In contrast, the method in [2] requires an additional 52% memory for reducing 41% of the write energy.

The rest of the paper is organized as follows. We describe the original weight reduction that was designed for asymmetric codes in Section 2 and show how the algorithm can be modified to reduce the number of '01' & '10' bit-patterns during ECC in Section 3. We prove that the error correction performance of the proposed method is the same as the original ECC in the Appendix. We describe the circuit-level overhead of implementing the proposed method in Section 4. Simulation results showing the write energy and latency savings as well as potential increase in endurance are presented in Sections 5 and 6. The paper is concluded in Section 7.

Following are the definitions and notations which will be frequently used in the next few sections:

Definition 1. If $\mathbf{c_n}$ is an n-bit codeword of a code C, then *weight of a codeword*, $w(\mathbf{c_n})$, is the number of 1s in $\mathbf{c_n}$.

Definition 2. t-EC (n, m, d_{min}) code stands for a symmetric ECC which can correct t errors in an n-bit codeword with minimum Hamming distance of d_{min}, where $d_{min} \geq 2t+1$. The number of information bits is m.

2. BACKGROUND: WEIGHT-REDUCTION ALGORITHM

In this section, we describe the *Weight-reduction algorithm* for reducing the *average weight* of a code. This algorithm was originally proposed in [6] for reducing weight in the design of asymmetric codes. We modify and apply this algorithm to reduce the write energy and latency for multi-level Flash memories.

2.1 Encoding

The block diagram of the encoder is given in Figure 1. The input to the encoder is a k-bit information vector $\mathbf{u_k}$ which is appended with bit '0' called the *Inverting bit* to generate the $(k+1)$-bit vector $\mathbf{u'_{k+1}}$. The vector $\mathbf{u'_{k+1}}$ is input to a t-EC $(n, k+1, d_{min})$ code which produces $\mathbf{c_n}$, an n-bit codeword. $\mathbf{c_n}$ is input to the *Weight-comparator block* which inverts the bit-vector $\mathbf{c_n}$ if the following condition is satisfied: If the weight of the codeword, $w(\mathbf{c_n})$ is greater than half the number of bits in the codeword $\mathbf{c_n}$, i.e., if $w(\mathbf{c_n}) > \frac{n}{2}$, the bit-vector $\mathbf{c_n}$ is inverted, i.e., $\mathbf{c'_n} = \mathbf{c_n} \oplus \mathbf{f_n}$, where $\mathbf{f_n}$ is the n-bit vector $\{11\ldots1\}$. Otherwise, if $w(\mathbf{c_n}) \leq \frac{n}{2}$, $\mathbf{c'_n}$ is the same as $\mathbf{c_n}$.

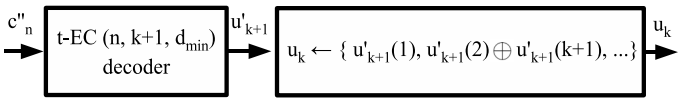

Figure 2: Block diagram of decoder for *Weight-reduction algorithm.*

2.2 Decoding

The block diagram for the corresponding decoder is given in Figure 2. The input is an n-bit vector $\mathbf{c''_n}$ which is fed to the decoder of t-EC $(n, k+1, d_{min})$ ECC. $\mathbf{c''_n}$ could be different from $\mathbf{c'_n}$ because of errors in the system. The decoder outputs a $(k+1)$-bit vector $\mathbf{u'_{k+1}}$, whose $(k+1)^{th}$ bit contains the information about the conditional inversion of codeword $\mathbf{c_n}$. The original k-bit information vector $\mathbf{u_k}$ can be obtained by doing binary bit-wise XOR of the first k bits with the $(k+1)^{th}$ bit of $\mathbf{u'_{k+1}}$. In other words,

$$\mathbf{u_k} = \{u'_{k+1}(1) \oplus u'_{k+1}(k+1), \ldots, u'_{k+1}(k) \oplus u'_{k+1}(k+1)\}$$

3. APPLICATION TO FLASH MEMORIES

In this section, we show how the *Weight-reduction algorithm* in Section 2 can be used to reduce the number of '01' & '10' bit-patterns in the data that is written in Flash memories. This algorithm is built on top of existing ECC in Flash memories. We describe the encoding and decoding process followed by simulation results comparing its error performance with that of the original ECC.

3.1 Encoding

The block diagram of the encoder for the modified *Weight-reduction algorithm* is given in Figure 3. The first two blocks are the same as in Figure 1. In addition, here a $\frac{n}{2}$-bit vector $\mathbf{x_{n/2}}$ is created, where $x_{n/2}(i) = c_n(2i-1) \oplus c_n(2i)$, $1 \leq i \leq \frac{n}{2}$. Thus a '1' in the vector $\mathbf{x_{n/2}}$ corresponds to a '01' or '10' bit-pattern in $\mathbf{c_n}$. The modified *Weight-reduction algorithm* conditionally modifies $\mathbf{c_n}$ if the number of 1s in $w(\mathbf{x_{n/2}})$ is $> \frac{n}{4}$. The modification involves bit-wise XOR of $\mathbf{c_n}$ with a constant vector $\mathbf{g_n}$, where $\mathbf{g_n}$ is a codeword of t-EC $(n, k+1, d_{min})$ code. $\mathbf{g_n}$ corresponds to the k-bit input $\{0101\ldots01\}$, when k is odd and $\{1010\ldots10\}$, when k is even. This choice of $\mathbf{g_n}$ guarantees that $\mathbf{c'_n}$ will always contain less number of '01' & '10' bit-patterns than $\mathbf{c_n}$. It also sets the $(k+1)^{th}$ bit of $\mathbf{c'_n}$ to '1' if there is a modification. The $(k+1)^{th}$ bit can be used to decode the modified vector back at the decoder.

3.1.1 Procedure

Step-1: Let $\mathbf{u_k}$ be the input vector. Appending bit-0 gives the $(k+1)$-bit vector $\mathbf{u_{k+1}}$.

Step-2: The t-EC $(n, k+1, d_{min})$ ECC gives an n-bit codeword $\mathbf{c_n}$ corresponding to the input vector $\mathbf{u_{k+1}}$.

Step-3: Compute $\mathbf{x_{n/2}}$: $x_{n/2}(i) = c_n(2i-1) \oplus c_n(2i)$, $1 \leq i \leq \frac{n}{2}$.

Step-4: If $w(\mathbf{x_{n/2}}) > \frac{n}{4}$ then $\mathbf{c'_n} = \mathbf{c_n} \oplus \mathbf{g_n}$, else $\mathbf{c'_n} = \mathbf{c_n}$. $\mathbf{g_n}$ is a codeword which corresponds to the input $\{0101\ldots01\}$, when k is odd, and $\{1010\ldots10\}$, when k is even.

3.1.2 Example

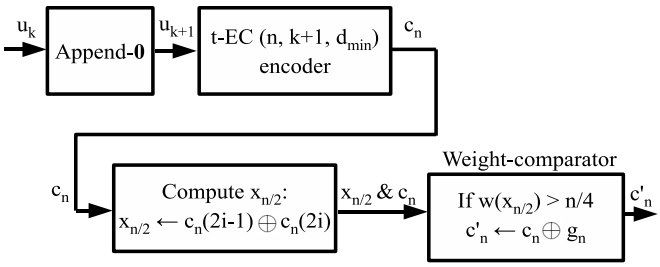

Figure 3: Block diagram of the encoder for MLC Flash memories.

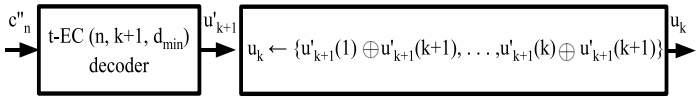

Figure 4: Block diagram of the decoder for MLC Flash memories.

Step-1: Let $\mathbf{u_3} = \{101\}$ be the input vector. Appending zero gives $\mathbf{u_4} = \{1010\}$.

Step-2: The codeword $\mathbf{c_8}$ is generated using 2-EC $(8, 4, 4)$ extended Hamming code, $\mathbf{c_8} = \{10101010\}$.

Step-3: $\mathbf{x_4}$ is $\{1111\}$.

Step-4: Since $w(\mathbf{x_4}) = 4 > \frac{8}{4}$, $\mathbf{c'_8} = \mathbf{c_8} \oplus \mathbf{g_8}$. Since $k = 3$, $\mathbf{g_8}$ is the codeword $\{01010101\}$ of 2-EC $(8, 4, 4)$ extended Hamming code corresponding to the input $\{0101\}$. Note that $\mathbf{c'_8} = \{10101010\} \oplus \{01010101\} = \{11111111\}$ has only '11' bit-patterns, which is significantly less costly to write in Flash memory.

3.2 Decoding

The block diagram of the encoder for the modified *Weight-reduction algorithm* is given in Figure 4. The n-bit vector $\mathbf{c''_n}$ is decoded to $\mathbf{u'_{k+1}}$ using the decoder for t-EC $(n, k+1, d_{min})$ code. The $(k+1)^{th}$ bit of the bit-vector $\mathbf{u'_{k+1}}$ has the information about the conditional modification at the encoder. When k is odd, the original k-bit information vector $\mathbf{u_k}$ is obtained by computing the XOR of every 2^{nd} bit with the $(k+1)^{th}$ bit, starting with the 2^{nd} bit. This is because at the encoder, the codeword $\mathbf{g_n}$ corresponds to the information vector $\{0101\ldots01\}$. When k is even, $\mathbf{g_n}$ corresponds to the information vector $\{1010\ldots10\}$, and the original k-bit information vector $\mathbf{u_k}$ is obtained by computing the XOR of every 2^{nd} bit with the $(k+1)^{th}$ bit, starting with the 1^{st} bit.

3.2.1 Procedure

Step-1: Let $\mathbf{c''_n}$ be the vector read from the memory. Decoding using the t-EC $(n, k+1, d_{min})$ code gives the error corrected vector $\mathbf{u'_{k+1}}$.

Step-2: Compute $\mathbf{u_k}$: When k is odd,
$\mathbf{u_k} = \{u'_{k+1}(1),\ u'_{k+1}(2) \oplus u'_{k+1}(k+1), u'_{k+1}(3), \ldots, u'_{k+1}(k)\}$.
When k is even,
$\mathbf{u_k} = \{u'_{k+1}(1) \oplus u'_{k+1}(k+1), u'_{k+1}(2), u'_{k+1}(3) \oplus u'_{k+1}(k+1), \ldots, u'_{k+1}(k) \oplus u'_{k+1}(k+1)\}$.

3.2.2 Example

Step-1: Let the vector read from the memory be $\mathbf{c''_8} = \{11111011\}$ with an error in the 6^{th} bit. The 2-EC $(8, 4, 4)$ extended Hamming code decoder corrects the error in the 6^{th} bit and outputs $\mathbf{u'_4} = \{1111\}$.

Step-2: $\mathbf{u_3} = \{u'_4(1), u'_4(2) \oplus u'_4(4), u'_4(3)\} = \{101\}$.

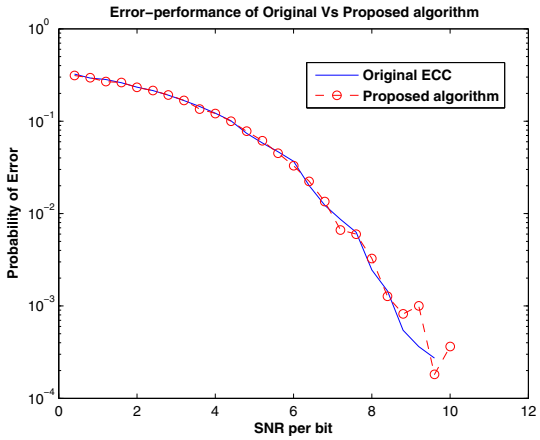

Figure 5: Error performance of the original ECC and the proposed modified *Weight-reduction algorithm.*

3.3 Error Performance

We show through simulations that the error performance of the proposed modified *Weight-reduction algorithm* is the same as that of original ECC. Specifically, we used a (15, 11, 3) Hamming code as the original ECC. Monte Carlo simulations for 10^5 iterations were done in MATLAB. Gaussian random noise was used for generating errors. The algorithm's error performance with SNR per bit is compared with that of the original ECC. The error performance of the original ECC and that of the proposed algorithm are given in Figure 5. SNR per bit is $20log\frac{Energy\ per\ bit}{var}$, where var is the variance of the Gaussian random noise. We see that the error performance of the proposed algorithm is exactly the same as that of the original ECC. This is formally proved in the Appendix.

4. IMPLEMENTATION ISSUES

In this section, we discuss the overhead involved in implementing the modified *Weight-reduction algorithm* in the ECC block.

4.1 Implementation of Encoder

From Figure 3, we see that the encoder for the modified *Weight-reduction algorithm* has four blocks. Out of these, the t-EC $(n, k+1, d_{min})$ encoder block for ECC already exists in the write interface of the Flash memory, so only the first, third and the fourth blocks account for the overhead.

The first block, *Append-0* adds '0' bit to the input. The implementation costs in terms of delay, area and power for this block is negligible. The third block calculates $x_{n/2}$ from $\mathbf{c_n}$ and has an overhead of $\frac{n}{2}$-XOR gates. The fourth block which checks $if\ w(\mathbf{x_{n/2}}) > \frac{n}{4}$ contributes the most to the overhead. It requires a counting circuit which counts the number of 1s in the vector $\mathbf{x_{n/2}}$. The circuit outputs logic 1 if the number of 1s are greater than the number of 0s and is referred to as majority voter. There are digital and analog

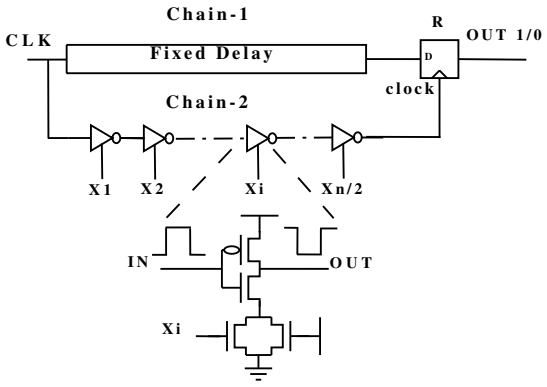

Figure 6: DMV circuit in the Weight-comparator.

majority voters in the literature. We focus our study on the digital majority voter (DMV).

4.1.1 Digital Majority Voter (DMV)

We propose a new DMV, the circuit of which is given in Figure 6. The circuit is based on a design presented in [7]. The complicated dynamic latching of the original design has been replaced by a simple register. The basic operation of the proposed DMV in Figure 6 is as follows. The input CLK signal is delayed through two paths. One path is through a chain of variable-delay inverters whose delay is controlled by the i^{th} bit of the vector $\mathbf{x_{n/2}}$. The other path has fixed delay; here it is chosen as the delay if $\frac{n}{4}$ '1's were fed to the chain of variable-delay inverters.

To understand the working of the proposed circuit, let us assume Chain-1 has more delay than Chain-2. Then at register R, the rising edge of CLK (through Chain-1) at input D arrives later than the rising edge of CLK (through Chain-2) at the clock input of register R. That means the rising edge of input D of R is shifted right with respect to the rising edge of CLK. So, in this case R will sample a '0'. If Chain-2 has more delay than Chain-1, R will sample a '1'.

Two-stage DMV implementation: For a MLC NOR Flash memory with block size 512 bits [2], the number of parity bits required is $2\text{xlog}\,512 = 18bits$ according to method in [8], and the codeword $n = 512 + 18 = 530$ bits. Thus, we need a DMV of size 265. A 265 long chain is not very realistic, so we propose breaking the DMV into two stages. All two stage implementations result in some loss in accuracy. We considered the following decompositions: 1) Sixty six 4-bit DMVs in the first stage followed by a 67-bit DMV in the second stage, 2) thirty three 8-bit DMVs in the first stage followed by a 34-bit DMV in the second stage, and 3) Sixteen 16-bit DMVs in the first stage followed by a 25-bit DMV in the second stage. Of these, the second combination has the performance closest to a single stage 265-bit DMV. Thus for the NOR flash with $k + 1 = 512$ bits, this combination gave incorrect outputs ≈5% of the time. For NAND flash with $k+1 = 4096$ bits, $n = 4096+24 = 4120$ bits, and a two-stage DMV with thirty two 64-bit DMVs in the first stage followed by a 44-bit DMV in the second stage gave the best result with ≈7% loss in accuracy.

The proposed DMV circuit includes a flip-flop and so metastability issues can arise. We propose using the 'metastability detector' in Razor flip-flop design [9] to detect cases when the input signal transition occurs close to the clock edge. For such cases, the DMV or the flip-flop's output can be taken to be 1 or 0. This does not affect the performance of the weight reduction algorithm much since metastability occurs (if at all) when the weight of the input vector is close to $\frac{n}{4}$.

Table 2: Latency and energy overhead of DMV implementations for various sized delay chains.

Size	Delay	Energy
8	0.8 ns	97 fJ
16	0.12 ns	163 fJ
32	0.26 ps	228 fJ
48	0.41 ps	294 fJ
64	0.60 ps	341 fJ

Table 3: Latency and energy overhead of the blocks in the $n = 530$, $k + 1 = 512$ encoder.

	Delay	Energy
First block	0 ps	0 pJ
Third block	11 ps	0.834 pJ
Fourth block	340 ps	2.7 pJ

4.1.2 Simulations

The simulations for the encoder blocks are done using HSPICE. Bulk High-performance (HP) 45 nm Predictive Technology Model (PTM) was used. Table 2 gives delay and energy overhead for the DMV implementations.

The total energy and delay overheads for the three blocks for $n = 530$, $k+1 = 512$ are tabulated in Table 3. The delay of the two-stage DMV (thirty three 8-bit DMVs followed by a 34-bit DMV) is $340ps$. Thus, the total delay and energy overheads are $351ps$ and $3.534pJ$ respectively. These low overheads make the implementation of the modified *Weight-reduction algorithm* a very attractive method.

The reliability of the proposed DMV circuits was also tested. The circuits were able to tolerate a ±15% variation from nominal 1.0 V and ±10% variation in transistor length for the 45 nm technology node.

4.2 Implementation of Decoder

In the decoder, the t-EC $(n, k+1, d_{min})$ Hamming decoder block already exists in the read interface of the Flash memory. The overhead is only due to the second block in Figure 4 which computes XOR of $(k+1)^{th}$ bit with every other bit. So the decoder overhead is due to $\frac{k}{2}$ parallel XOR gates. Simulations show that the total delay and energy overheads are $11ps$ and $256fJ$ respectively, for $k + 1 = 512$.

5. SAVINGS IN ENERGY AND LATENCY

In this section, we first walk through an example for calculating write energy and latency savings. We then give detailed simulation results of the write energy and latency savings of the algorithm for various bit-streams derived from real computing sources.

5.1 Illustrative example

Let {101110100101} be the 12 bits that have to be written into Flash memory. Let the ECC be 2-EC $(8, 4, 4)$ extended Hamming code.

Table 4: Energy and latency savings for various bit-streams for Intel MLC NOR 28F256L18 Flash memory with $n = 530$, $k + 1 = 512$. In the notation A(B), A corresponds to the proposed two-stage DMV implementation and B corresponds to the ideal single-stage DMV implementation.

Bit-stream	Size	Energy savings	Latency savings
Mp3	10 MB	33% (39%)	32% (38%)
Linux	699 MB	35% (39%)	33% (37%)
Adobe PDF	2 MB	31% (36%)	30% (34%)
Images (.jpg)	5.6 MB	32% (37%)	31% (35%)
Speech (.wav)	8 MB	36% (42%)	35% (41%)
Gaussian	100 MB	30% (38%)	28% (36%)

In the traditional case, the bit-stream is divided into 3 blocks with 4 bits per block. The 3 blocks: {1011}, {1010} and {0101} are encoded by the (8,4,4) extended Hamming code to give codewords {10110100}, {10101010} and {01010101} respectively. Note that there is one '00', five '01', five '10' and one '11' bit-patterns. Using Table 1 for NOR Flash memory, the write energy required is $1 \times 4.738 + 5 \times 29.531 + 5 \times 31.194 + 1 \times 0.752 = 309.115\ \mu J$ and the write latency is $1 \times 110.00 + 5 \times 644.23 + 5 \times 684.57 + 1 \times 24.93 = 6778.93\ \mu s$.

If the proposed modified *weight reduction algorithm* is to be applied, the bit-stream is divided into 4 blocks with 3 bits per block to accommodate for the inversion bit in each block. The 4 blocks: {101}, {110}, {100} and {101} are appended with 0s to give {1010}, {1100}, {1000} and {1010} respectively, and encoded by the extended Hamming code to give codewords {10101010}, {11001100}, {10000111} and {10101010} respectively. The *Weight-comparator block* conditionally XORs the codewords with $g_8 = \{01010101\}$. In this case, only the first and the fourth vectors were XORed and the final vectors are {11111111}, {11001100}, {10000111} and {11111111}. Since there are three '00', one '01', one '10' and eleven '11' bit-patterns to be written into memory, using Table 1, the write energy required is $83.211\ \mu J$ and the write latency is $1933\ \mu s$. In this particular example, the proposed modified *weight reduction algorithm* reduces write energy and latency by 73% and 71% respectively.

5.2 Examples from real bit-streams

Industry standards for ECCs used in NAND Flash memories are described in [8], [10] and [11]. Typically NOR Flash memories do not have on-chip ECC but MLC NOR Flash now demand use of strong ECC protection due to higher probability of error [12], [11]. We consider the Hamming code as the ECC to demonstrate the write energy and latency savings in case of NOR Flash memory. The block size of the Hamming code is chosen as $k + 1 = 512$ bits.

Bit-streams from real computing sources have been chosen as input bit-streams. These include compressed files such a regular mp3 file of size 10 MB, Ubuntu-8.10 Linux zImage file of size 699 MB, an Adobe PDF of size 5.6 MB and four 256x256 JPEG images. We have also used an uncompressed .wav file and a bitstream of size 100 MB generated from concatenating 32-bit numbers that are randomly drawn from a Gaussian distribution.

We will discuss the case with mp3 file as bit-stream. Our simulations show that without the modified *Weight-reduction algorithm*, the mp3 file requires writing of 63 '00', 71 '01', 74 '10' and 57 '11' bit-patterns per 530 bits on an average over 10 MB. The write energy is 4.71 mJ and latency is 104 ms. Using the modified *Weight-reduction algorithm*, the mp3 file requires writing of roughly 90 '00', 43 '01', 42 '10' and 95 '11' bit patterns per 530 bits on an average. The write energy is 3.16 mJ and latency is 70.72 ms per 530 bits resulting in significant savings.

Table 5: Reduction in number of 01 & 10 bit-patterns for NAND Flash memory with $k+1 = 4096$. In the notation A(B), A corresponds to the proposed two-stage DMV implementation and B corresponds to single-stage DMV implementation.

Bit-stream	Size	Reduction
Mp3	10 MB	49% (54%)
Linux	699 MB	48% (54%)
Adobe PDF	2 MB	45% (50%)
Images (.jpg)	5.6 MB	47% (51%)
Speech (.wav)	8 MB	51% (55%)
Gaussian	100 MB	47% (51%)

The write energy and latency savings for Intel MLC NOR 28F256L18 Flash memory are tabulated in Table 4. The difference between two-stage and single-stage DMV implementation is due to loss of accuracy as described in Section 4.1.1. We note a loss of just 5% savings due to using the two-stage DMV. Simulations show similar results for other Flash memories like Samsung MLC NOR KAK38200AM.

Next we present results for NAND Flash memory with block size of 4096 bits [8], [10]. Table 5 gives the percentage reduction in 01s & 10s for the two-stage and single-stage DMV implementations. We did not have access to NAND Flash energy and latency measurement values and could not provide the savings results. However, we expect the savings to be significant for NAND Flash memories as well.

6. IMPROVEMENT IN ENDURANCE

In a 4-level NOR Flash memory, of the four states, '11' is called erased state since it does not require programming the memory cell, while '00', '01' & '10' are called programmed states. The endurance of a Flash memory cell strongly depends upon the number of programming operations on the cell [13], [14]. The use of modified *Weight-reduction algorithm* reduces the number of '01' & '10' bit-patterns which translates to reduction in the number of program cycles and improvement in the endurance of the Flash memory cells. Table 6 lists the average improvement in endurance for the different bit-streams. Improvement in endurance is calculated as the reduction in the average number of program cycles. We conclude from Table 6 that the endurance or life time of Flash memory cells increases by nearly 24%.

7. CONCLUSION

The energy and latency for writing data in Flash memories are an order of magnitude higher than traditional memories. Moreover, multi-level Flash memories suffer from endurance and reliability issues. In this paper, we design an algorithm to reduce the average write energy and latency in Flash memories as well as improve the endurance. The algorithm is built on top of existing ECC in Flash memories and effectively reduces the number of '01' & '10' bit-patterns in

Table 6: Improvement in endurance for the various bit-streams.

Bit-stream	Size	Improvement in endurance
Mp3	10 MB	26%
Linux	699 MB	25%
Adobe PDF	2 MB	22%
Images (.jpg)	5.6 MB	22%
Speech (.wav)	8 MB	28%
Gaussian	100 MB	24%

the codeword that is written into the memory. The overhead of this algorithm is contributed primarily by a digital majority voter (DMV). The use of the proposed modified *Weight-reduction algorithm* saves 33% and 31% of write energy and latency, respectively, for Intel NOR Flash memories. It also improves the endurance of the Flash memory by 24%.

ACKNOWLEDGEMENTS

This work was supported in part by a grant from the National Science Foundation CSR-EHS 0615135.

REFERENCES

[1] T. Kgil, D. Roberts, and T. Mudge, "Improving NAND flash based disk caches," *35th International Symposium on Computer Architecture*, pp. 327–38, 2008.

[2] Y. Joo and et. al., "An energy characterization platform for memory devices and energy-aware data compression for multilevel-cell Flash memory," *ACM Transactions on Design Automation of Electronic Systems*, vol. 13, July 2008.

[3] S. Lin and D. J. Costello, *Error Control Coding*. Prentice Hall, 2 ed., 2004.

[4] S. Gregori and et. al., "On-chip error correcting techniques for new-generation Flash memories," *Proceedings of the IEEE*, vol. 91, April 2003.

[5] B. Chen, X. Zhang, and Z. Wang, "Error correction for multi-level NAND flash memory using Reed-Solomon codes," *IEEE Workshop on Signal Processing Systems*, pp. 94–99, November 2008.

[6] J. Bruck and M. Blaum, "Some new EC/AUED Codes," *The Nineteenth International Symposium on Fault-Tolerant Computing*, pp. 208–15, Nov 1989.

[7] M. Fujino and V. Moshnyaga, "An efficient Hamming distance comparator for low-power applications," *9th International Conference on Electronics, Circuits and Systems*, vol. 2, pp. 641–644, September 2002.

[8] "Hamming codes for NAND Flash memory devices," *http://www.micron.com/products/nand/technotes/*.

[9] D. Ernst and N. S. Kim, "Razor: A Low-Power Pipeline Based on Circuit-Level Timing Speculation," *36th Annual International Symposium on Microarchitecture*, pp. 7–18, Dec 2003.

[10] S. Chen, "What types of ECC Should be used on Flash memory?," *Application Note, http://www.eetasia.com.*

[11] F. Sun and et. al., "Design of on-chip error correction systems for multilevel NOR and NAND Flash memories," *IET, Circuits, Devices and Systems*, vol. 1, pp. 241–249, June 2007.

[12] B. Godard and et. al., "Hierarchical Code Correction and Reliability Management in Embedded NOR Flash memories," *European Test Symposium*, pp. 84–90, May 2008.

[13] "IEEE Standard Definitions and Characterization of Floating Gate Semiconductor Arrays," Feb 1999.

[14] N. Mielke and et. al., "Bit error rate in NAND Flash memories," *IEEE International Reliability Physics Symposium*, pp. 9 – 19, April-May 2008.

APPENDIX:PROOF OF ERROR-CORRECTION CAPABILITY

In this section, we prove that the proposed modified *Weight-reduction algorithm* does not affect the error correction capability of the ECC. Specifically, we show that the minimum Hamming distance d_{min} of the proposed system is the same as the original ECC.

PROOF. Let t-EC $(n, k+1, d_{min})$ be a t-error correcting code $\mathbf{C}$ used in step-2 of the modified *Weight-reduction algorithm*. The minimum Hamming distance $d_{min} \geq 2t+1$ by definition. The input vector in step-2 of the algorithm is the vector $\mathbf{u_{k+1}} = \{\mathbf{u_k}, 0\}$, where $\mathbf{u_k}$ corresponds to the k information bits. Since $\mathbf{C}$ is a code with minimum Hamming distance d_{min}, all the codewords in the code $\mathbf{C}$, except the all-0 vector $\mathbf{0_n}$, have a minimum weight of at least d_{min},

$$w(\mathbf{c_n}) > d_{min} \text{ for any codeword } \mathbf{c_n} \in \mathbf{C}$$

In the encoding algorithm, $\mathbf{g_n}$ is a codeword of $\mathbf{C}$ corresponding to k-bit input $\{0101\ldots01\}$ when k is odd and corresponding to the input $\{1010\ldots10\}$ when k is even. Step-4 of the encoding algorithm is given by

$$if\ w(\mathbf{x_{n/2}}) > \frac{n}{4}\ then\ \mathbf{c'_n} = \mathbf{c_n} \oplus \mathbf{g_n}\ else\ \mathbf{c'_n} = \mathbf{c_n}$$

Alternately, $if\ w(\mathbf{x_{n/2}}) > \frac{n}{4}\ then\ \mathbf{c'_n} = \mathbf{c_n} \oplus b.\mathbf{g_n}$, where b is 1 or 0 depending on condition $w(\mathbf{x_{n/2}}) > \frac{n}{4}$, and $b.\mathbf{g_n}$ is obtained by 'AND'ing bit b with each of the bits in the vector $\mathbf{g_n}$.

Let $\mathbf{y'_n}$ & $\mathbf{z'_n}$ be any two output vectors corresponding to the any two input vectors $\mathbf{y_n}$ & $\mathbf{z_n} \in \mathbf{C}$. The Hamming distance between the two output vectors $\mathbf{y'_n}$ and $\mathbf{z'_n}$ can be written as

$$d(\mathbf{y'_n}, \mathbf{z'_n}) = d(\mathbf{y_n} \oplus e.\mathbf{g_n}, \mathbf{z_n} \oplus f.\mathbf{g_n})$$

where e and f are 1 if the number of '01' & '10' bit-patterns is larger than $\frac{n}{4}$ in $\mathbf{y_n}$ and $\mathbf{z_n}$ respectively. Since the Hamming distance between any two codewords $\mathbf{p_n}, \mathbf{q_n}$ of a code is the weight of the codeword $\mathbf{p_n} \oplus \mathbf{q_n}$, $d(\mathbf{y'_n}, \mathbf{z'_n})$ can be expressed as

$$d(\mathbf{y'_n}, \mathbf{z'_n}) = w(\mathbf{y_n} \oplus e.\mathbf{g_n} \oplus \mathbf{z_n} \oplus f.\mathbf{g_n})$$

Since $\mathbf{r_n} = \mathbf{y_n} \oplus e.\mathbf{g_n} \oplus \mathbf{z_n} \oplus f.\mathbf{g_n}$ is a linear combination of the codewords of the code $\mathbf{C}$ in the Galois Field $\mathbf{GF(2)}$, the resultant vector $\mathbf{r_n}$ is also a codeword of the code $\mathbf{C}$. By definition, $w(\mathbf{r_n}) > d_{min}$. Thus,

$$d(\mathbf{y'_n}, \mathbf{z'_n}) > d_{min}$$

Thus, the minimum Hamming distance between any two output vectors of the encoder is d_{min}, and the error correcting capability remains the same as the original ECC. □