In the *46th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*
Toulouse, France, June 2016

# Methuselah Flash: Rewriting Codes for Extra Long Storage Lifetime

Georgios Mappouras, Alireza Vahid, Robert Calderbank, Daniel J. Sorin
Department of Electrical and Computer Engineering
Duke University
{georgios.mappouras, alireza.vahid, robert.calderbank, sorin}@duke.edu

***Abstract*—Motivated by embedded systems and datacenters that require long-life components, we extend the lifetime of Flash memory using rewriting codes that allow for multiple writes to a page before it needs to be erased. Although researchers have previously explored rewriting codes for this purpose, we make two significant contributions beyond prior work. First, we remove the assumption of idealized—and unrealistically optimistic—Flash cells used in prior work on endurance codes. Unfortunately, current Flash technology has a non-ideal interface, due to its underlying physical design, and does not, for example, allow all seemingly possible increases in a cell's level. We show how to provide the ideal multi-level cell interface, by developing a virtual Flash cell, and we evaluate its impact on existing endurance codes. Our second contribution is our development of novel endurance codes, called Methuselah Flash Codes (MFC), that provide better cost/lifetime trade-offs than previously studied codes.**

***Keywords—Flash memory; Coding; Lifetime; Endurance***

## I. INTRODUCTION

Flash memory is being increasingly used, due to its increasing capacity and the narrowing of the cost differential between Flash and other storage technologies (especially hard drives). NAND Flash is the dominant technology of solid-state drives (SSDs) and numerous other storage devices. Typical storage devices use multi-level cells with 2 (SLC), 4 (MLC) or 8 (TLC) levels per cell. MLCs and TLCs are usually preferred because they provide better storage density than SLCs.

One drawback to using Flash is that its cells wear out after a number of program/erase (P/E) cycles. That is, we can only erase a Flash cell a given number of times before that cell can no longer retain information. The number of P/E cycles that a cell can tolerate is the lifetime of the cell and it depends on the type of the cell used (SLC, MLC or TLC) and the Flash technology node size. The node size is decreasing rapidly as Flash cells continue to shrink at each generation in order to provide greater density. However smaller node sizes can endure fewer P/E cycles.

We seek to improve Flash's endurance and, in doing so, it is important to understand when and where endurance needs to be improved. Solid state drives (SSDs) in typical personal computers are an example where lifetime extension is unnecessary, because the expected lifetime of an SSD exceeds the 3-5 year lifetime of the computer itself. Because coding techniques to extend the lifetime incur a cost—in terms of the extra raw capacity required to provide a given amount of host-visible capacity—we do not wish to pay that cost unnecessarily.

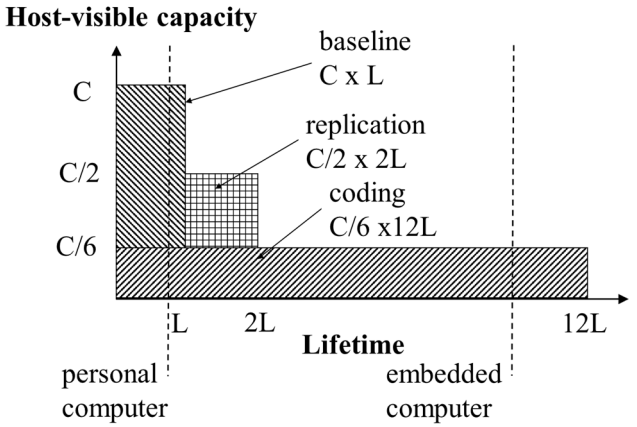


Figure 1. Host-visible capacity as function of lifetime, with fixed cost (in raw capacity).

In Figure 1, we illustrate the trade-off between lifetime and cost in the context of a baseline that is today's Flash with no modifications to extend its lifetime. The x-axis is lifetime, normalized to $L$, the lifetime of the baseline. The y-axis is host-visible capacity, normalized to $C$, the host-visible capacity of the baseline. The figure contains rectangles which represent equal-cost (in terms of raw capacity) trade-offs between lifetime and host-visible capacity. The baseline has a rectangle of $C$ host-visible capacity at $L$ lifetime, which has the same cost as the replication scheme ($C/2$ at $2L$) and a coding scheme ($C/6$ at $12L$) we describe later. Note that equal cost does not necessarily imply equal rectangle area.

Figure 1 also contains two dotted vertical lines that denote target lifetimes for different applications. These dotted lines are not meant to represent exact lifetimes for specific applications, but rather to illustrate big-picture differences in application lifetime needs. For example, we draw the dotted line for personal computers to the left of $L$ on the x-axis, meaning that the baseline Flash suffices (This line could shift to the right over time, as $L$ decreases in each technology generation). The dotted line for certain embedded systems that require long life (e.g., space probes, embedded sensor platforms, etc.) and SSDs in datacenters is far to the right of $L$ and requires a lifetime extension scheme even if it incurs a reduction in host-visible capacity (at the same cost as the baseline) and/or an increase in cost (to achieve the same host-visible capacity as the baseline).

For those scenarios in which Flash lifetime needs to be extended, there are two primary and largely complementary approaches: endurance codes and wear-leveling. In this work, we focus on endurance codes, in which we encode datawords

to codewords before writing them to the Flash, so as to extend the Flash's lifetime.

We make two contributions in this work. First, we bridge the gap between the idealized—and unrealistically optimistic—Flash cells used in the prior work on endurance codes. Prior endurance codes show promise [1, 2, 3, 4, 5, 6, 7], in theory, but they are incompatible with the current Flash interface. The codes expect "ideal" multi-level cells, in which each cell has some number of levels, $L$, and each cell can be increased from level $i$ to level $j$ as long as $i<j$. Unfortunately, current Flash technology has a non-ideal interface, due to its underlying physical design, and does not, for example, allow all seemingly possible increases in a cell's level. In this paper, we show how to provide the ideal multi-level cell interface, by developing what we refer to as a virtual Flash cell, and we evaluate its impact on existing endurance codes. We demonstrate how to create virtual cells with any number of levels independently of the Flash type and technology used.

Our second contribution is our development of novel endurance codes, called *Methuselah Flash Codes (MFC)*, that provide far better cost/lifetime trade-offs than previously studied codes. We start with the general concept of coset coding [8, 9], in which each dataword to be written maps to a unique coset of codewords. Coset coding provides multiple options for which codeword to write, and our contribution is the development of new heuristics for choosing codewords from cosets so as to maximize the lifetime of Flash.

## II. Flash Background

In this section, we describe how NAND Flash is organized and operates, we discuss its endurance issues, and we explain some important details of its interface that have a large impact on coding. We consider only NAND Flash, because of its ubiquity, and we use the term Flash to refer to it.

### A. *Flash memory organization*

A Flash chip consists of some number of blocks, where each block contains some number (128-256) of pages. Page sizes are typically on the order of 4-16KB, and pages are the smallest units in Flash that can be read or written. Blocks are the smallest units that can be erased, and thus a block erase causes many pages to be erased at the same time.

To minimize block erases—which is important for endurance, as we discuss later—Flash updates are not performed "in place." A write to update a page of data already on the chip is performed to a clean page, and the page that held the previous data is marked as invalid. The Flash Translation Layer (FTL) software maintains the mapping from each logical page to the location of its most recently updated data, and it also performs garbage collection to free up blocks with many invalid pages. To free a block, the FTL copies out any valid pages to new free pages (in another block) and then erases the block.

### B. *Flash Cells and Wearout*

Flash SSDs consist of NAND Flash cells, and each cell can be interpreted as having two or more distinct levels. Flash chips are often classified based on whether the cells are interpreted as storing one bit per cell (SLCs), 2 bits per cell (MLCs) or 3 bits per cell (TLCs). The name "single-level cell (SLC)" is a historical misnomer; an SLC actually has two levels, 0 and 1. Also, while MLC stands for multi-level cell it actually refers to a cell of 4 levels. TLCs (triple level cells) refer to 8 level cells.

Without loss of generality, we will assume MLCs (i.e., each cell has 4 levels L0, L1, L2, and L3) in this discussion, for the purposes of making the examples and explanations concrete.

Writing to a Flash cell involves adding charge to the cell. The amount of charge depends on the level desired; that is, more charge is required when changing the level from L0 to L2 than when changing the level from L0 to L1. Erasing a cell removes all of its charge and sets its level to L0. (There is no way to decrease a cell's level except for erasing back to level L0.) We assume that the Flash cells support what is known as "program without erase" (PWE), i.e., a cell's value can be changed without being erased first, as long as the value is being incremented [10]. We have experimentally tested that we can perform PWE on reasonably modern Flash chips, a Samsung K9LCG08U1M and a Hynix H27QDG8VEBIR SK.

Flash cells can be erased only so many times before they wear out (i.e., cannot be written again), and this is the fundamental problem we address in this work. Mohan et al. [11] show how Flash cells may recover from wearout, to some extent, but the fundamental problem of wearout remains.

### C. *Important Issues in Flash Interface*

The interface provided by Flash chips has two important—and often overlooked—quirks that impact how one might develop coding techniques for Flash.

First, one might expect that the level of an MLC can be increased arbitrarily. That is, one might expect to be able to change a MLC's level from L0 to L1, L2, or L3, from L1 to L2 or L3, and from L2 to L3. Unfortunately, this is not the case. At the physical level, a MLC's level can be changed from L0 to L1 or L2 (but not L3), from L1 to L3 (but not L2), and from L2 to L3. Furthermore, the interface provided to Flash does not even provide access to cells of any kind; rather, the interface currently provided by the FTL software is simply pages of bits. A Flash chip can be accessed by reading/writing bits on pages, but not by reading/modifying levels of cells. A code that assumes the naïve interface—ideal multi-level cells—will not work on today's Flash chips.

Second, one might expect that a single MLC represents two bits on a given page. However, that is not the interface provided by today's chips. Instead a single MLC represents one bit on one page (let us name that "page x") and one bit on another page (let us name that "page y") in the same block. Once again, a code that assumes the naïve interface will not work on today's Flash chips.

Figure 2 describes those limitations schematically. We observe that a transition from L1 to L2 implies that a bit in page x should flip from 1 to 0. This however is not a legal transition and the FTL will not allow it to occur. Additionally the transition from L0 to L3 cannot be performed in a single program request as that would require programing both pages x and y.

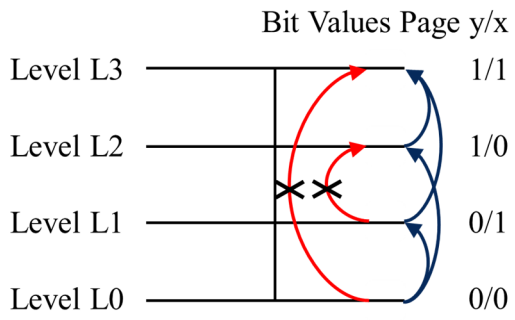


Figure 2. An MLC with its allowed transitions.

Although some researchers have identified these interface quirks [12, 13], we are unaware of any prior work on rewriting codes that accounts for these quirks.

## III. Theory of Endurance Coding

Methuselah Flash builds upon a coding technique developed by Jacobvitz et al. [6]. We start by describing a technique known as waterfall coding [14], which Jacobvitz et al. use to connect the concept of coset coding with the use of multi-level cells. All of the work described in this section is prior work; our contributions are in the next three sections, where we apply this prior theory to realistic Flash.

### A. *Waterfall Coding*

We assume, for purposes of explanation, that pages consist of ideal 4-level cells. However, instead of using the 4 levels of the cell to hold 2 bits of data (as is typical), waterfall coding [14] uses the 4 levels to hold 1 bit of data, as illustrated in Figure 3. Levels L0 and L2 correspond to a bit value of 0, and Levels L1 and L3 correspond to a bit value of 1. An erased cell is at Level L0 (bit value 0). Subsequent writes to the cell add charge to it to increase its level. Thus, an erased cell can progress from bit value 0 (L0) to 1 (L1) to 0 (L2) and back to 1 (L3). At Level L3, the cell is saturated with charge and may not be programmed again, so a subsequent write to change the bit value to 0 requires the cell to be erased. Using a 4-level cell in this way enables a single cell to be written multiple times before it needs to be erased. Throughout this paper, we leverage waterfall coding when we consider (virtual) cells with more than 2 levels.

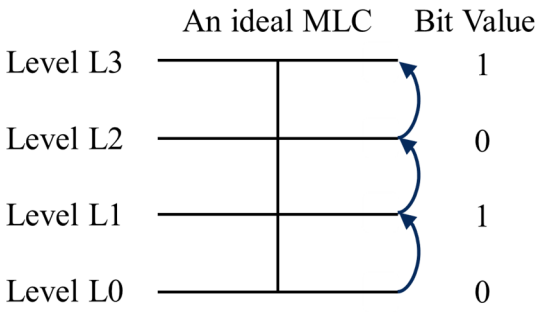


Figure 3. Waterfall Coding for ideal MLCs.

### B. *Write Once Memory (WOM) codes*

The idea of reusing a "write-once" memory was first presented by Rivest and Shamir [15]. Since then WOM codes (and variations of them) have been extensively used in order to enhance Flash's lifetime [2, 3, 4, 5, 7, 16, 17]. The general idea is to represent a number of bits ($b$) with a number of multi-level cells ($m$), where each cells has a number of levels ($L$). By increasing the level of one or more of the $m$ different cells you can re-program the $b$ bits to a different value.

How the different sequences of $b$-bits are mapped to the $L^m$ possible values of the cells depends on the specific implementation. A simple example of a WOM code is presented in Figure 4, where 3 2-level cells (ovals) are used to store 2 bits (values above or below the ovals). This WOM code enables two writes before erasing.

### C. *Coset Coding for Endurance*

In this section we present the basic idea of coset codes and how they can be used in order to increase Flash lifetime. We also demonstrate how coset codes are generated, focusing on the particular coset codes that are the basis for Methuselah Flash Codes.

#### *1) Using Cosets*

Consider a single Flash page to be written, and assume the page-sized dataword to be stored is $X$. In typical storage systems, there is a one-to-one mapping between $X$ and the codeword that is actually written, which we denote as $Y$. $Y$ could, for example, be $X$ augmented with parity bits in an error correcting code (ECC).

The key feature of coset coding is that it changes the model from a one-to-one mapping to a one-to-many mapping. Consider a system with $k$-bit datawords and $n$-bit ($n=k+c$) codewords. With coset coding, we divide the $n$-bit space into equal sized cosets. We perform a one-to-one mapping of each $k$-bit dataword to a coset, i.e., we have $2^k$ cosets, each with $E=2^c$ codewords. That is, for any given dataword $X$, there are $E$ possible codewords $\{Y_1 \ldots Y_E\}$ that we can write. There is a one-to-one mapping from a dataword to coset but a one-to-many mapping from a dataword to possible codewords.

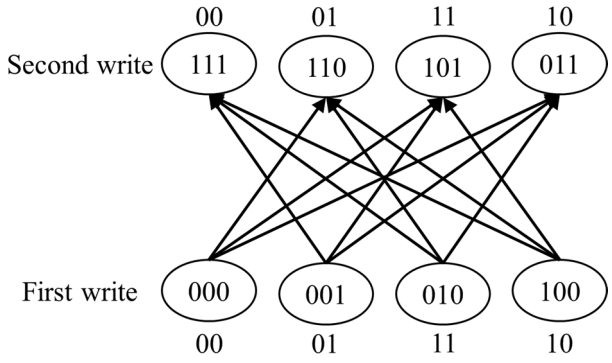


Figure 4. A WOM code example. Writing two bits twice in three 2-level cells.

Benefit: We choose the codeword that optimizes an objective, and our high-level objective in this work is postponing wearout. In Section V.A, we precisely state the concrete objectives that enable us to postpone wearout.

Cost: The cost of coset coding is its overhead for representing a codeword. Each codeword has $n=k+c$ bits, and the extra $c$ bits are the overhead for the code. A code's cost is often referred to as its *rate*, which is defined as the dataword size divided by codeword size.

*2) Generating Cosets*

The key to coset coding is coset generation, which is the process of dividing the codeword space into $2^k$ cosets, where each coset has $E=2^c$ codewords. Coset generation is performed using a code, and there are many codes that can be used for this process. These codes include block codes and convolutional codes. Different codes offer different trade-offs between overhead (i.e., how many extra bits are required to represent a codeword compared to a dataword) and flexibility (i.e., how many options are in each coset). The drawback with block codes is the difficulty in matching bit patterns across block boundaries. However, convolutional codes have no block boundaries except at the beginning and at the end and therefore convolutional codes are more suitable for our work. In this paper, we use convolutional codes of rate 1/2, 1/3, 1/4, and 1/5 to create coset codes of rate 1/2, 2/3, 3/4, and 4/5, respectively.

There are various characteristics that define a convolutional code, such as rate and number of states. As mentioned above, we consider various rates in this work. We also considered multiple rate-1/2 convolutional codes with different number of states. Increasing the number of states in the state machine provides a bigger set of codewords to choose from; therefore allowing greater benefits to be achieved. These greater benefits come at the cost of negligibly lower rates.

All of the information required to construct the codes we use can be found in Table 12.1 (c) of Lin and Costello's textbook [18].

## IV. Virtualizing Flash Cells

Our first goal is to bridge the gap between the interface provided by the FTL software in Flash memories and the ideal multi-level cell interface assumed by prior work in coding.

The ideal interface—the interface assumed by most coding theorists and the interface we seek to provide with our virtual cells—provides the illusion of a cell with levels 0 to $L$-1, and each level can be increased from level $i$ to level $j$ as long as $i<j$. If a cell reaches level $L$-1, it can no longer be programed until the cell is erased (as part of the block being erased).

To achieve this ideal interface, we build on the existing interface that provides pages with bits. We provide a general solution that can be used to generate virtual cells (v-cells) with any number of levels, independent of the type of physical cells that are used in the Flash chip (SLC, MLC, or TLC). *Regardless of the technology, our approach to all v-cell designs remains the same: interpret the values of multiple bits of the same page as the levels of a single v-cell.*

Our approach overcomes the limitations imposed by the interface provided by Flash chips, which is pages of bits rather than cells with levels. Although there are other possibilities, we choose to implement the virtual cell interface by augmenting the FTL software, which serves as the bridge between the device driver software on the host computer and the Flash chips. As illustrated in Figure 5, we extend the FTL with the ability to perform coding on top of v-cells; extensions are shown as shaded software modules within the FTL. The software module that implements v-cells provides the v-cell interface that supports independently written coding modules. None of these changes to the FTL are visible to the host computer.

We now present two examples of virtual MLCs that we have developed.

### A. *Example 1: A 4-Level Virtual Cell*

In order to create each 4-level v-cell, we group three consecutive bits of a page. The level of the v-cell is determined by counting how many of the three bits are at a value of 1. Thus a v-cell in level L0 has its three bits at value 000. A v-cell in level L1 has its three bits at value 001, 010, or 100, a v-cell in level L2 has its bits at 011, 101, or 110, and a v-cell in level L3 has its bits at 111. We illustrate this v-cell and the mappings from levels to bits in Figure 6. Because some levels have multiple bit representations, we can transition between them in different ways.

We can now use this v-cell as an ideal MLC. We can choose to store one or more bits in it. We can also implement

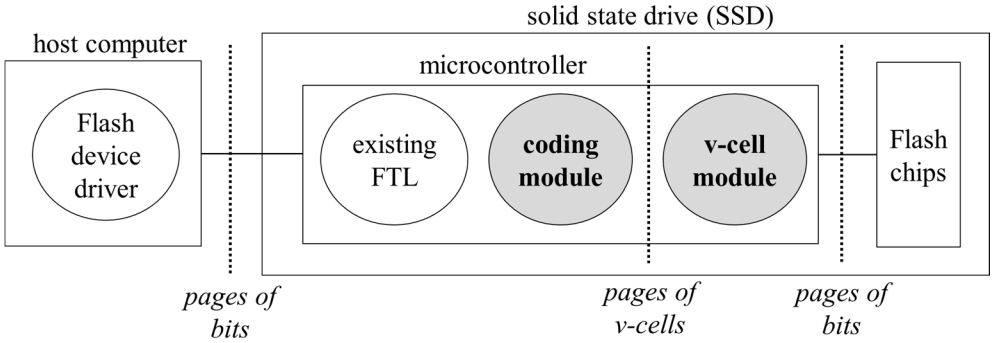


Figure 5. Implementing the virtual cell interface by extending the FTL software.

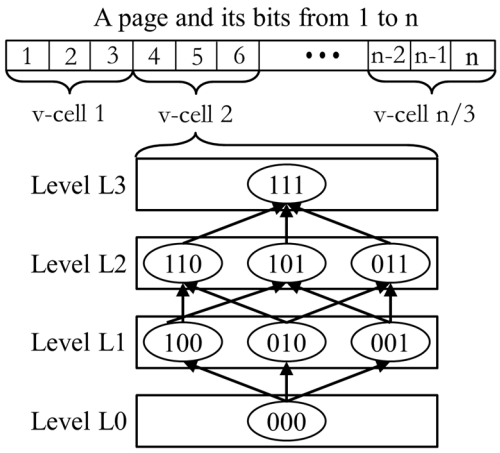


Figure 6. Using the v-cell technique to create an ideal 4-level cell.

waterfall codes, WOM codes, coset codes or any other code on top of it.

### B. *Example 2: A 8-Level Virtual Cell*

To create an 8-level v-cell we need to group the bits in sequences of 7 bits. Notice that any $L$-level v-cell can be generated by grouping $L$-1 bits together. This grouping provides us with a "bigger" cell that is shown in Figure 7.

For simplicity we do not show all the transitions and all the representations for each level. However one can reason about them by using the same procedure as we did for creating the ideal 4-level cell.

## V. Methuselah Flash

Methuselah Flash Codes (MFCs) build upon the theory of coset coding. Our key innovation in this work—beyond developing the v-cells that facilitate coding—is developing heuristics for choosing codewords in cosets so as to provide the best Flash endurance. A MFC is a coset code that uses our heuristics. In our evaluation later in this paper, we experiment with different coset codes, but we use the same codeword selection algorithms. Without loss of generality, we assume that all MFCs are implemented on top of ideal 4-level v-cells.

### A. *Codeword Selection Objectives*

With coset coding, a dataword maps to a coset of codewords, and we can select any codeword from that coset to write. We now use three examples, shown in Figure 8, to illustrate our three objectives in this selection process. Figure 8(a) shows the initial value of a page with 12 4-level v-cells and we use this same initial value in all three examples. The numbers indicate the level of each v-cell.

**(1) Avoid Codewords that Increment Saturated Cells.** Example 1, in Figure 8(b), shows two possible options for which cells to increment, assuming that we have used coset coding to provide two possible codewords, $Y_i$ and $Y_j$, for each dataword $X$. The top option, $Y_i$, is unwriteable, because it requires an increment of a cell (shaded in the figure) that is already at level L3. The bottom option, $Y_j$, does not increment that cell and thus choosing it postpones the need to erase. Intuitively, our goal is to avoid incrementing cells at L3 and, in turn, to avoid incrementing cells to L3 if other cells can be incremented instead.

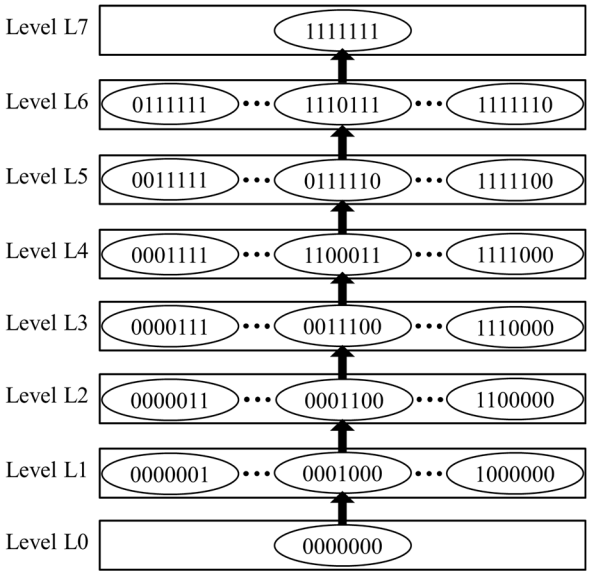


Figure 7. Using the v-cell technique to create an ideal 8-level cell.

**(2) Minimize the Number of Cells Incremented.** Example 2, in Figure 8(c), shows another two options for writing. In this example, $Y_j$ is preferable to $Y_i$ because it increments fewer cells and thus, all other things being equal, postpones erasing for longer.

**(3) Balance Increments Across Cells.** Example 3, in Figure 8(d), shows two more options for writing, where both increment the same number of cells. Despite this seeming

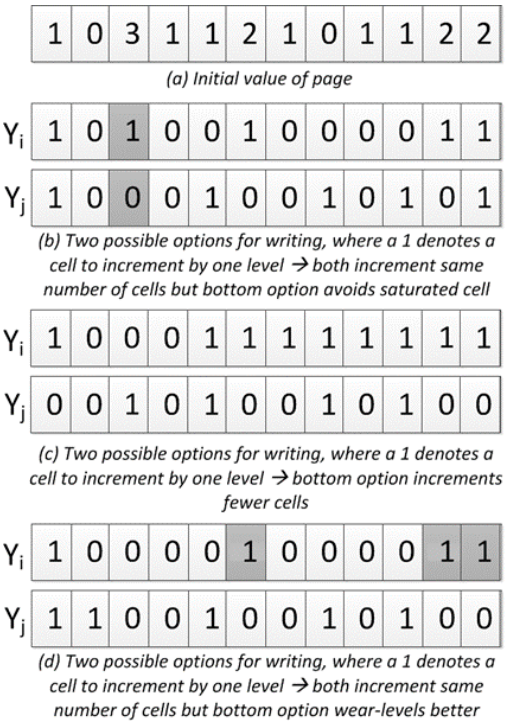


Figure 8. Codeword Selection Examples

equivalence, $Y_j$ is preferable to $Y_i$ because it balances the way that cells are incremented better; the top option increments cells at L2 (shaded), whereas the bottom option preferentially increments cells at L0 or L1. The bottom option is thus better at postponing erasing.

MFCs integrate these three objectives into a unified metric function that determines a cost for each possible codeword that we could write. This metric function is used by the Viterbi algorithm to search a coset and decide which codeword achieves the best (minimum) cost, i.e., performs better in all of these three objectives.

The overall goal is to minimize the cost of writing to a page, which is the sum of the costs of writing to each cell in that page.

$$cost_{page} = \sum_{i=1}^{n} cost_{cell_i}$$

The cost to write a given cell in a page is a function of the current level of the cell ($l$), the level to which it would need to be increased ($l'$), and the maximum number of levels in the cell ($L$).

$$cost_{cell_i} = f(l, l', L)$$

The function $f$ considers our three objectives. First, we avoid codewords that increment saturated cells by setting $f=\infty$ for saturated cells (i.e., $l=L-1$). Second, we minimize the number of cells incremented by setting $f=0$ for cells that do not need to be programmed (i.e., $l=l'$). Third, we balance increments by setting $f=l'$ and thus favoring cell writes with lower post-write levels.

$$f(l, l', L) = \begin{cases} 0, & l = l' \\ \infty, & l \neq l' AND\ l = L-1 \\ l', & l \neq l' AND\ l < L-1 \end{cases}$$

The Viterbi algorithm efficiently decides which of the candidate codewords achieves the minimum cost. Note that the current level of each v-cell, as well as the maximum number of levels in the v-cells, are independent of the codewords. However each codeword leads to different post-write cell levels and thus a different cost.

### B. *Integration with Error Correction*

Flash chips are expected to tolerate errors in cells. Transient and permanent errors can affect cells, and current Flash standards require the ability to correct at least one error per 1024 cells. For example, current SSDs use error correcting codes (ECC) for this purpose.

Although error correction is mostly complementary to our goals and not the focus of this paper, it is important to note that the coset coding technique we use [6] has already been shown to be compatible with error correction. The key idea is to ensure that cosets consist solely of valid ECC-protected codewords. We still have a mapping from a dataword $X$ to a coset of codewords $\{Y_1 \ldots Y_E\}$, and we can now ensure that all elements $Y_i$ are valid ECC-protected codewords.

To maintain the same number of elements per coset while providing error correction, we must increase the size of each codeword and thus decrease the rate of the code. Recall that each coset contains $E=2^c$ codewords. Without error correction, all $2^c$ $c$-bit vectors could belong to cosets, but to provide error correction we must discard those $c$-bit vectors that are not valid ECC-protected codewords. Thus we need a larger value of $c$ if we are not going to use all of the vectors in the space of $c$-bit vectors. In this way, adding error correction increases the storage cost of coset coding.

There are two considerations when choosing the specific ECC to use. First, the choice of ECC determines how many errors can be corrected (e.g., SECDED) and how much storage overhead is required for error correction. Second, the ECC must be "compatible" with the code used for coset generation.

It is important to note that a naive implementation of ECC fixes a division between information bits and parity bits. Schechter et al. [19] showed that such an implementation can hurt endurance (of PCM, but similar reasoning applies to Flash) because the ECC bits get flipped far more frequently than the data bits they protect. Indeed, if we simply computed ECC for each codeword and appended ECC to the codeword, we would hurt our potential lifetime gain as the cells that are used to store ECC bits will saturate way faster than the rest of the cells; however, when ECC is integrated with the coset code into a single code, we preserve all of the balancing properties of the coset code.

Because the focus of our work is on postponing wearout, rather than tolerating errors, we do not further consider ECC in this paper. We simply note that it is a complementary feature that could be added without affecting our MFC heuristics for choosing codewords within cosets. The MFCs we demonstrate and analyze assume no error protection.

## VI. Implementations

In this section, we present the implementations for different codes on top of the v-cell interface. We demonstrate MFCs as well as a WOM code. Although WOM codes are already extensively used in prior works, here we analyze their gains under a realistic implementation and use those results as a comparison point. We also remind the reader that all our implementations are based on the 4-level v-cells.

### A. *WOM*

For the WOM code, we use the 4-level v-cell to store two bits of data. Thus the implementation has an overall rate of 2/3, because each v-cell—which consists of three bits—holds only 2 bits. In other words, for a raw capacity of $C$ we have a host-visible capacity of $2/3C$.

In Figure 9 we demonstrate how the different representations of two bits are mapped to different states and levels of the v-cell. Note that some levels provide multiple options by taking advantage of the multiple paths between levels. Under each state we mark the bits stored at that state. We also demonstrate an example as we re-program the cell in Figure 9 in order to update its data. In the first re-program we transition from L0 to L1 by flipping the first bit of the 3-bit triplet representing the v-cell. Notice that by doing so the other two options of L1 become unreachable. That means we can only visit them if we first erase the v-cell. In this example the cell is updated 4 times before saturating.

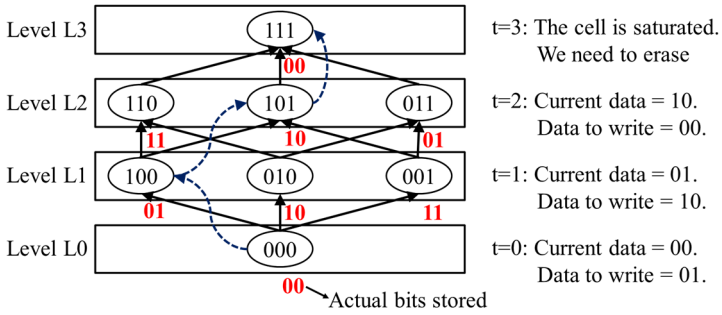


Figure 9. WOM and v-cell representation. Under each state we mark the bits that each state represents. The dashed arrows indicate how we transition from ine state to an other for the illustrated example.

## B. *Methusalah Flash Codes (MFCs)*

All MFCs in this work use the metric function that was presented in Section V.A. We present MFCs based on coset codes with multiple rates; 1/2 (MFC-1/2), 2/3 (MFC-2/3), 3/4 (MFC-3/4) and 4/5 (MFC-4/5).

For MFC-1/2, we use the v-cells in two different ways, by storing 1 bit per cell (MFC-1/2-1BPC) and 2 bits per cell (MFC-1/2-2BPC) as shown in Figure 10. These two options present a potential tradeoff between host visible capacity loss and lifetime gains. Notice that, although MFC-1/2 is based on a coset code with rate 1/2, the overall implementation has a rate 1/6 when using the v-cell with 1BPC and a rate of 1/3 when using the v-cell with 2BPC.

For the rest of the MFCs we only explore the case where each v-cell stores 1BPC. Thus the implementations of MFC-2/3, MFC-3/4 and MFC-4/5 have a rate of 2/9, 1/4 and 4/15 respectively.

## C. *Performance and Implementation Analysis*

Any implementation of a coding scheme incurs overheads in performance (latency and/or bandwidth) and energy. These overheads arise due to both the logic for encoding/decoding and the extra Flash accesses that may need to be performed. These overheads are not unique to MFCs or re-writing codes, but rather apply to any coding scheme that is used for any purpose.

In the context of Flash, which has relatively slow access times (compared to, say, SRAM or DRAM), the logic for encoding/decoding incurs relatively little overhead. Moreover, the performance impact of encoding/decoding can be mitigated with special-purpose hardware, if desired.

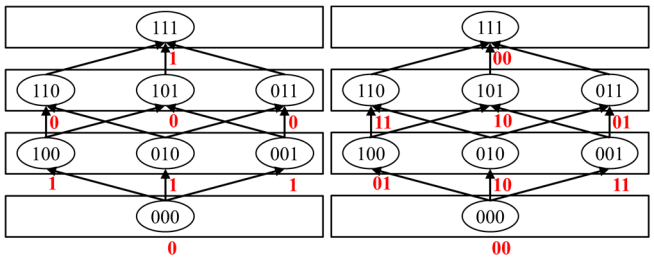


Figure 10. Two ways to map bits to cells for cosets.

The more challenging overhead arises due to the need for extra accesses to the Flash. A code with rate $r$ (say, 1/2) requires each Flash access to read or write $1/r$ times more bits than an uncoded Flash. If a user accesses one page of data, the implementation must access $1/r$ pages. The overhead of these extra accesses could be mitigated by exploiting parallelism within and across Flash chips, when possible. It is also possible that a custom Flash chip design, targeted for re-writing codes, could have larger page sizes in order to fit more data per page and thus require fewer reads/writes per access.

Coding overheads depend highly on both the code used and the details of the overall implementation. For example, a re-writing code could be implemented at different system levels (OS, drivers, or FTL) and could be accelerated with specialized hardware to reduce performance overhead.

These overheads are inherent to any coding scheme, and they are a necessary price to pay to extend lifetime. It is not the case that we can choose a lifetime extension scheme without overheads; rather, we decide how much lifetime extension we need and we then engineer the system to minimize the overheads as best we can.

# VII. Methodology

We now describe how we evaluate MFCs and quantitatively compare them to prior work.

## A. *Evaluation Metrics*

Our goal is to increase Flash lifetime while minimizing the cost of doing so. Any lifetime extension scheme has some cost, which is an increase in *raw capacity* and/or a decrease in *host-visible capacity*. Raw capacity is the capacity that would be visible if the Flash were used without any lifetime extension scheme. The host-visible capacity—the capacity visible to the user and the operating system for the whole life of the Flash product—is less than the raw capacity if a coding scheme is used. The *rate* of a code—which we previously defined as the size of a dataword divided by the size of a codeword—is thus also equal to the host-visible capacity divided by the raw capacity; an uncoded Flash has a rate of 1, and all coding schemes have rates less than 1.

The benefit of a lifetime extension scheme is its *lifetime gain*, which we measure as its number of program/erase (PE) cycles divided by the PE cycles of an uncoded Flash that allows a page to be programmed once before being erased.

Our key evaluation metric is the lifetime gain multiplied by the rate, and we refer to this metric as *aggregate gain*. An uncoded Flash has an aggregate gain defined equal to one, and our goal is to develop schemes with aggregate gains greater than one. Referring back to Figure 1, the areas of the rectangles for each scheme equal their aggregate gains.

We note that schemes with equal aggregate gains may not be equally desirable. For example, a scheme with lifetime gain 3 and rate 1/2 may be more practically useful than a scheme with lifetime gain 30 and rate 1/20, even though both have the same aggregate gain of 3/2.

### B. *Schemes Compared*

The baseline to which we compare is uncoded Flash with capacity $C$ (raw capacity equals host-visible capacity for the baseline) and lifetime $L$. As mentioned above, its aggregate gain is defined to equal 1.

We also compare to a simple redundancy scheme in which we use a factor of $K$ times as much raw capacity to achieve the same host-visible capacity $C$, for a rate of $1/K$. With simple redundancy, we use the first $C$ of the raw capacity, without coding, until it wears out. Then we use the next $C$ of raw capacity until it wears out, etc. Simple redundancy thus provides a lifetime gain of $K$. The aggregate gain is thus $K/K$=1, which is no better than the baseline.

We also evaluate the WOM code and the MFC codes described in Section VI.

### C. *Simulation*

We simulate a single 4KB page of Flash as it is repeatedly programmed by a stream of writes. We record the average number of writes that can be performed to this page before it needs to be erased, and this value is the lifetime gain. Because the WOM codes and MFCs effectively scramble the datawords in converting them to codewords, the results are independent of the input data that is written. For simplicity, we model the writes with pseudo-randomly generated data.

To fix the Flash page size, despite the codes having different rates, we vary the size of the datawords to be written, so that the codewords are all page-sized. That is, for a given implementation with rate $r$ we choose a dataword size, $d$, such that $d \times r$ = 4KB. Varying the dataword size is a reasonable approach because, even in modern uncoded Flash implementations, the data are grouped in the appropriate size before stored, so as to accommodate the possible difference in sizes between a Flash hardware page and a page of virtual memory.

## VIII. EVALUATION

Using the methodology described in the previous section, we determined the lifetime gain and aggregate gain for each lifetime extension scheme. These results are summarized in Table I, and they show that different implementations provide a wide range of trade-offs between cost (rate) and benefit (lifetime gain). In the rest of this section, we delve more deeply into these high-level results.

TABLE I. RATE, LIFETIME AND AGGREGATE GAIN FOR ALL THE IMPLEMENSTATION.

| Code | Rate | Lifetime Gain | Aggregate Gain |
|---|---|---|---|
| BASELINE | 1 | 1 | 1 |
| Redundancy of *K* | 1/*K* | *K* | 1 |
| WOM | 2/3 | 2 | 1.33 |
| MFC-1/2 – 2BPC | 1/3 | 4 | 1.33 |
| MFC-2/3 | 2/9 | 6.99 | 1.55 |
| MFC-3/4 | 1/4 | 6.26 | 1.56 |
| MFC-4/5 | 4/15 | 5.94 | 1.58 |
| MFC-1/2 – 1BPC | 1/6 | 12 | 2 |

### A. *Fixed-Cost Comparisons*

To highlight the differences between the implementations, we fix the raw capacity at $C$, the capacity of the baseline uncoded Flash, and show how each implementation provides a different trade-off between host-visible capacity and lifetime gain. We illustrate these trade-offs using figures similar to Figure 1, in which the x-axis is lifetime gain and the y-axis is host-visible capacity. The area of each rectangle represents aggregate gain.

In Figure 11, we show the advantages of MFCs, with respect to prior work, by comparing three MFCs with the baseline, redundancy, and a WOM code. We make three observations from this figure. First, MFCs (e.g., MFC-1/2) can achieve greater aggregate gains than redundancy or WOM. Second, an MFC can have the same aggregate gain as a WOM code while providing a different trade-off of host-visible capacity versus lifetime gain, as exemplified by MFC-1/2-2BPC and the WOM code in the figure. Third, two implementations that provide the same lifetime can provide different host-visible capacities, depending on their aggregate gain (WOM vs Redundancy-1/2)

In Figure 12, we compare all of the MFCs to each other. We observe that they offer a wide range of trade-offs. MFC-1/2-2BPC, MFC-2/3, MFC-3/4 and MFC-4/5 achieve a range of lifetime gains from 4 to almost 7. MFC-1/2-1BPC stands out from the rest of the MFCs with a remarkable lifetime gain of 12.

### B. *Cost to Achieve Extreme Lifetime*

To highlight the importance of aggregate gain, we consider a situation that demands extreme lifetime. We assume an application that requires a lifetime gain of 12 and we compare the cost (raw capacity) of different coding schemes, in order achieve that requirement, for different host-visible capacity goals.

Figure 13 summarizes these results for the WOM code, MFC-4/5 and MFC-1/2 codes, and redundancy. The results are normalized to a baseline of capacity $C$ and lifetime $L$. We observe that MFC-1/2, which has the largest aggregate gain, provides the cheapest solution in comparison to the other codes. From this graph, we conclude that higher aggregate gains provide cheaper solutions (in terms of raw capacity).

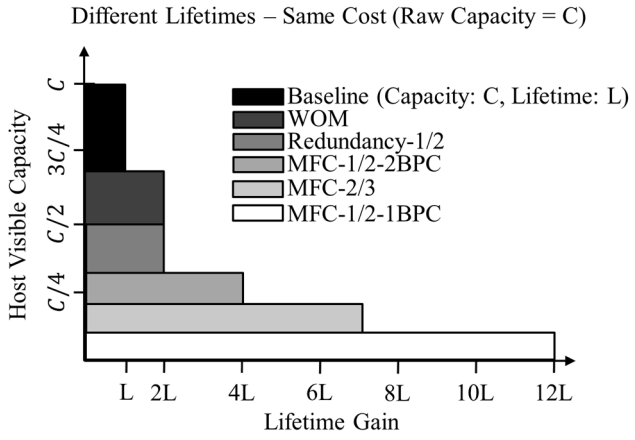


Figure 11. Fixed-cost comparisons of MFCs to prior work.

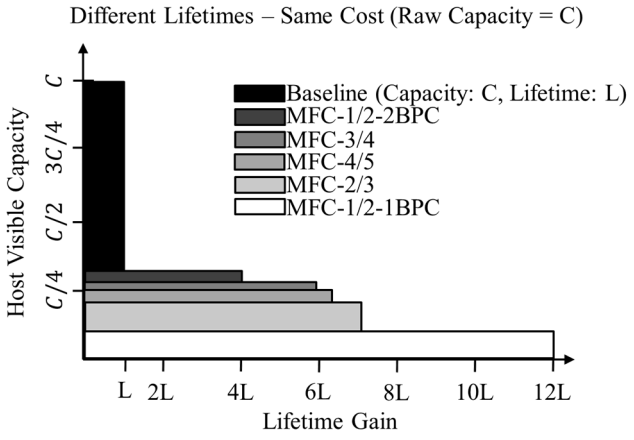


Figure 12. Fixed-cost comparisons of different MFCs.

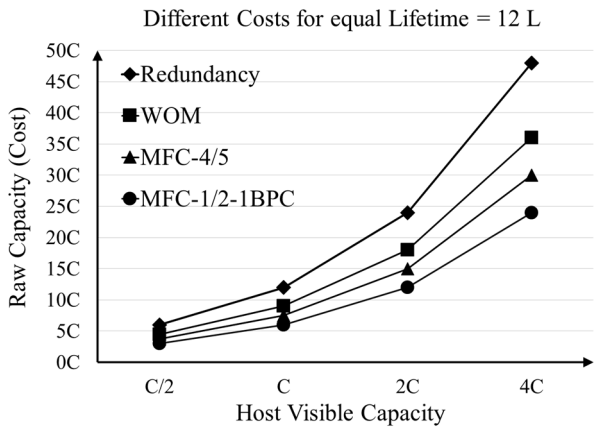


Figure 13. Different costs for a given lifetime and host visible capacity goal.

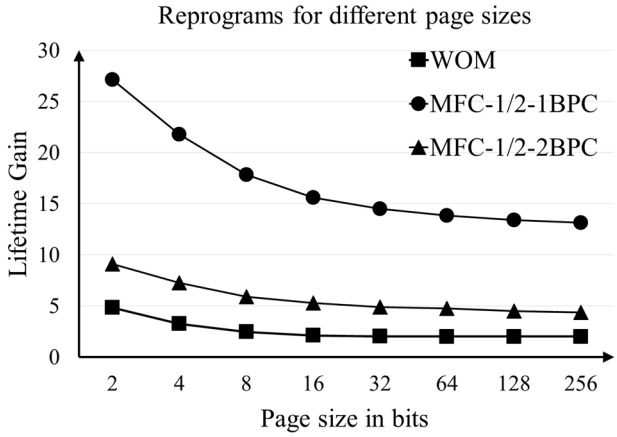


Figure 14. Lifetime gain for different page sizes.

## C. *Sensitivity Analysis: Lifetime vs Flash Page Size*

A code's ability to postpone erasing depends somewhat on the Flash page size. A page is no longer re-programmable after a sequence of input data that cause some of the cells to saturate. The number of re-programs before a cell becomes saturated varies depending on the sequence of bits that we want to store to that cell. Some input data sequences will cause cells to saturate faster than others. As the page size increases, the probability that such a "bad" sequence of inputs will occur for any of our cells increases. Thus it increases the probability of having saturated cells that will act as a bottleneck in our lifetime gain.

In Figure 14, we plot lifetime gain as a function of page size, for WOM and two MFCs. The results show that smaller page sizes indeed provide better lifetime gains. However, we cannot decide to have arbitrarily small Flash pages; there are implementation reasons for having reasonably large (multi-kilobyte) pages. For example, smaller pages require more metadata to track the mapping between data and pages and more complex garbage collection mechanisms.

## D. *Analysis of MFC Objectives*

MFCs choose codewords from cosets so as to achieve three objectives: avoid writing to saturated cells, minimize the number of cells that increment, and balance the increments across the cells. In this section, we show how well MFCs achieve the latter two objectives; the first objective is always required. The results in this section help to explain the higher level results presented earlier.

### *1) Minimize the Number of Cells that Increment*

For each page update, we calculate the fraction of cells that increment. We further distinguish these results based on how many updates have already been done to this page since it was last erased. We compare WOM and MFC-1/2-1BPC, so recall that the WOM code achieves only 2 updates per page while MFC-1/2-1BPC achieves 12 updates.

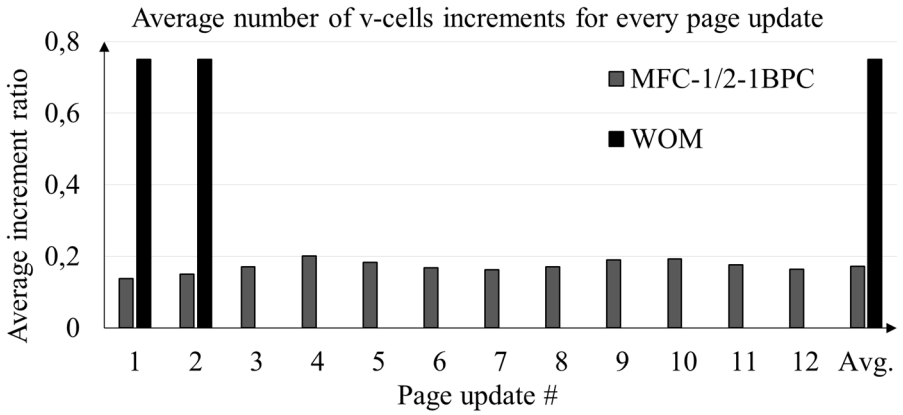


Figure 15. Average number of increments.

We show the results in Figure 15, in which the x-axis is the page update number since its last erase, and the y-axis is the average fraction of cells that increment. We also present, on the far right, an average over all page update numbers. We observe that MFC-1/2-1BPC has on average 17% of the v-cells incremented in each update, whereas WOM has an average of 75%. We also notice that in the case of MFC-1/2-1BPC the first two updates have the fewest increments (~14%). The reason we observe that is because, in the first two updates, the majority of the v-cells are in level L0 and thus the cost of balancing increments is minimal. In the later updates, the number of increments is increased as MFCs also try to balance increments.

*2) Balance Increments Across Cells*

We calculate the histogram of the levels that the cells reach before the page gets erased. We present that result in Figure 16 for MFC-1/2-1BPC and the WOM code.

We observe that for the case of MFC-1/2-1BPC the majority of the cells reach level L2 and, on average, only 0.5% of the v-cells stay on level L0. That means that 99.95% of the v-cells are programmed at least once while 88.5% of them reach level L2 or L3. In an ideal case, all cells would have reached level L3 before erase, but that is not achievable.

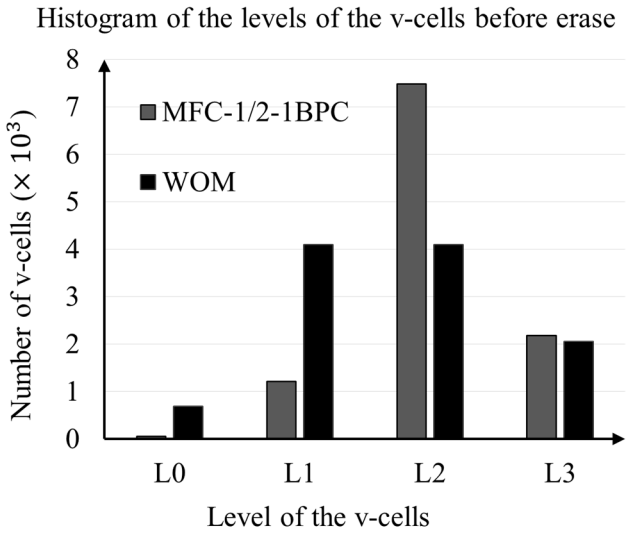


Figure 16. Histogram of the v-cell levels before erasure.

In the case of the WOM code, only 56% of the v-cells reach levels L2 or L3 and 6% of them never get programmed. Interestingly both implementations have about the same number of v-cells in level L3 (~20%). That result shows that saturated cells are a crucial bottleneck for re-writing codes and indicates that 20% of the v-cells being saturated is an average number that causes the whole page to be unable to be re-programmed.

## IX. Related Work

There are two complementary approaches for extending lifetime: postponing wearout and tolerating wearout. Our MFCs, described in Section V, focus on postponing wearout and can be combined with ECC, as shown in Section V.B, to also tolerate wearout.

### A. *Postponing Wearout*

There are two techniques for postponing wearout: coding and wear-leveling. Since we have already discussed coding, we focus here on wear-leveling.

Intuitively, one would prefer not to wear out one or a handful of cells out of the thousands of cells in a page, thus rendering the entire page unusable. At a larger scale, one would prefer not to wear out one or a handful of pages out of the many pages in a block, thus reducing the effective size of the block (leading to more frequent erasing of the block). It is important that all cells across pages and blocks are wearing out uniformly in order to ensure a good lifetime performance.

Many schemes have been developed for wear-leveling at different granularities. The main focus on Flash memories is on the block granularity [20, 21, 22]. By adding some extra complexity in the FTL algorithm, blocks can have a more even number of erases. Problems arise with blocks containing "cold" and "hot" data, meaning data that are either rarely or frequently modified respectively. These algorithms try to detect such blocks and evenly distribute the erases by using various techniques. This may increase the overall number of block erasures.

Flash memory, in its current form, does not require page wear leveling mechanisms as every page inside a block is only programmed exactly one time before erased. However in the

case of a coding scheme, like MFCs, wear leveling at that granularity may be beneficial. Research in that area [23] has provided solutions for other memory types (i.e. PCM). The main idea is to select the pages to program in such an order, so that you evenly use all pages

### B. *Tolerating Wearout*

Tolerating wearout is an important aspect in Flash memory for multiple reasons. The first reason is that some Flash cells may start wearing out a lot faster than the indicated lifetime due to various defects in the cells. Additionally some cells may be manufactured with defects and thus be unable to retain information from the start of the life of the product. Such failures are commonly observed in Flash memories [13].

For these reasons common Flash memory implementations provide some extra capacity that is used for ECC [13]. Different ECCs have been proposed [24, 25, 26] that explore tradeoffs between complexity, size and correction capabilities.

Other ways of tolerating wearout, besides ECC, have been also proposed. Schechter et al. [19] use a finite number of redundant cells that are used to replace the initial defective cells as well as the cells that wear out faster than expected. Although their scheme is presented for PCM it can be used for any non-volatile memory.

## X. Conclusion

We conclude that Methuselah Flash Codes can provide several benefits for Flash. We showed that MFCs achieve the best aggregate gains in comparison to prior work, as well as providing a range of trade-offs between rate and lifetime gain for a given aggregate gain. Furthermore, we believe that the compatibility of MFCs with ECC makes MFCs particularly attractive.

We also conclude that, regardless of the coding scheme, one must carefully consider the interface provided by a realistic Flash. Rather than assuming idealized cells, we highlighted the limitations of the current interface and found a way to provide virtual cells that facilitate coding on real Flash.

Another conclusion of this work is that there could be benefits to co-designing Flash chips with code designers and systems designers. Decisions like the mapping of cell levels to bits and the sizes of the pages could be optimized for a given purpose and maximize the benefits of re-writing codes like MFCs. Eslami et al. [27] showed how such a co-design process could be beneficial for phase change memory (PCM), and it is possible that co-design for Flash offers similar opportunities.

## Acknowledgment

This material is based on work supported by the National Science Foundation under grant CCF-142-1177.

## References

[1] A. Jiang, R. Mateescu, M. Schwartz and J. Bruck, "Rank Modulation for Flash Memories," IEEE Transactions on Information Theory, vol.55, no.6, pp.2659-2673, June 2009.

[2] W. Chua, K. Cai and Wang Ling Goh, "Efficient Two-Write WOM-Codes for Non-Volatile Memories," IEEE Communications Letters, vol.19, no.10, pp.1690-1693, Oct. 2015.

[3] A. Bhatia, M. Qin, A. Iyengar, B. Kurkoski and P. Siegel, "Lattice-Based WOM Codes for Multilevel Flash Memories," IEEE Journal on Selected Areas in Communications, vol.32, no.5, pp.933-945, May 2014.

[4] A. Jiang, V. Bohossian and J. Bruck, "Floating Codes for Joint Information Storage in Write Asymmetric Memories," IEEE International Symposium on Information Theory, pp.1166-1170, Jun. 2007.

[5] S. Kayser, E. Yaakobi, P. Siegel, A. Vardy and J. Wolf, "Multiple-write WOM-codes," in 48th Annual Allerton Conference on Communication, Control, and Computing, pp.1062-1068, Oct. 2010.

[6] A. Jacobvitz, R. Calderbank and D. Sorin, "Writing Cosets of a Convolutional Code to Increase the Lifetime of Flash Memory," in 50th Annual Allerton Conference Communication, Control, and Computing, pp. 308–318, 2012.

[7] B. Kurkoski, "Rewriting Flash Memories and Dirty-paper Coding," IEEE International Conference on Communications, pp.4353-4357, Jun. 2013.

[8] G. J. Forney, "Coset Codes. I. Introduction and Geometrical Classification," IEEE Transactions on Information Theory, vol.34, no.5, pp.1123-1151, Sep. 1988.

[9] G. J. Forney, "Coset Codes. II. Binary Lattices and Related Codes," IEEE Transactions on Information Theory, vol.34, no.5, pp.1152-1187, Sep 1988.

[10] R. Hasbun and F. Janecek, "Multiple Writes Per a Single Erase for a Nonvolatile Memory," U.S. Patent No. 5,936,884. 10 Aug. 1999.

[11] V. Mohan, T. Siddiqua, S. Gurumurthi, and M. R. Stan, "How I Learned to Stop Worrying and Love Flash Endurance," in Proceedings of the 2nd USENIX Conference on Hot topics in Storage and File Systems, pp. 3–3, 2010.

[12] Y. Cai, O. Multu, E. F. Haratsch, K. Mai, "Program Interference in MLC NAND Flash Memory: Characterization, Modeling, and Mitigation," IEEE 31st International Conference on Computer Design (ICCD), pp.123-130, 2013.

[13] L. Grupp, A. Caulfield, J. Coburn, S. Swanson, E. Yaakobi, P. Siegel and J. Wolf, "Characterizing Flash Memory: Anomalies, Observations, and Applications," in 42nd Annual IEEE/ACM International Symposium on Microarchitecture, pp.24-33, Dec. 2009.

[14] L. A. Lastras-Montaño, M. Franceschini, T. Mittelholzer, J. Karidis, and M. Wegman, "On the Lifetime of Multilevel Memories," in Proceedings of the 2009 IEEE International Symposium on Information Theory, vol. 2, pp. 1224–1228, 2009.

[15] R. L. Rivest and A. Shamir, "How to Reuse a "Write-once" Memory," Information and Control, vol.55, no.1, pp.1-19, 1982.

[16] E. Yaakobi, S. Kayser, P. Siegel, A. Vardy and J. Wolf, "Efficient Two-write WOM-codes," in Information Theory Workshop, pp.1-5, 2010.

[17] E. Yaakobi, S. Kayser, P. Siegel, A. Vardy and J. Wolf, "Codes for Write-Once Memories," IEEE Transactions on Information Theory, vol.58, no.9, pp.5985-5999, Sept. 2012.

[18] S. Lin and D. J. Costello, Jr, Error Control Coding, 2nd ed. Pearson Prentice Hall, 2004.

[19] S. Schechter, G. H. Loh, K. Straus and D. Burger, "Use ECP, not ECC, for Hard Failures in Resistive Memories," in Proceedings of the 37th Annual International Symposium on Computer Architecture, pp.141-152, 2010.

[20] L.-P. Chang, "On Efficient Wear Leveling for Large-scale Flash-memory Storage Systems," in Proceedings of the 2007 ACM Symposium on Applied Computing (SAC '07), pp.1126-1130, ACM, 2007.

[21] Y.-H. Chang, J.-W. Hsieh and T.-W. Kuo, "Endurance Enhancement of Flash-memory Storage Systems: An Efficient Static Wear Leveling Design," in Proceedings of the 44th Annual Design Automation Conference, pp.212-217, Jun 2007.

[22] L.-P. Chang and C.-D. Du, "Design and Implementation of an Efficient Wear-leveling Algorithm for Solid-state-disk Microcontrollers,"ACM Transactions on Design Automation of Electronic Systems (TODAES), vol. 15, no. 1, pp.1-36, 2009.

[23] M. K. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras and B. Abali, "Enhancing Lifetime and Security of PCM-based Main Memory with Start-gap Wear Leveling," in Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture, pp.14-23, 2009.

[24] G. Dong, N. Xie and Tong Zhang, "On the Use of Soft-Decision Error-Correction Codes in NAND Flash Memory," IEEE Transactions on Circuits and Systems I: Regular Papers, vol.58, no.2, pp.429-439, Feb. 2011.

[25] B. Chen, X. Zhang and Zhongfeng Wang, "Error Correction for Multi-level NAND Flash Memory Using Reed-Solomon Codes," IEEE Workshop on Signal Processing Systems, pp.94-99, Oct. 2008.

[26] S. Gregori, A. Cabrini, O. Khouri and G. Torelli, "On-chip Error Correcting Techniques for New-generation Flash Memories," in Proceedings of the IEEE , vol.91, no.4, pp.602-616, April 2003.

[27] A. Eslami, A. Velasco, A. Vahid, G. Mappouras, R. Calderbank and D. J. Sorin, "Writing without Disturb on Phase Change Memories by Integrating Coding and Layout Design," in Proceedings of the 2015 International Symposium on Memory Systems, ACM, pp.71-77, 2015.