Extending Flash Lifetime in Embedded Processors by Expanding Analog Choice

Georgios Mapporuas, *Member*, *IEEE,* Alireza Vahid, *Member*, *IEEE,* Robert Calderbank, *Fellow*, *IEEE,* and Daniel J. Sorin, *Senior Member*, *IEEE*

ABSTRACT

We extend the lifetime of Flash memory in embedded processors by exploiting the fact that data from sensors is inherently analog. Prior work in the computer architecture community has assumed that all data is digital and has overlooked the opportunities available when working with analog data, such as the data recorded by sensors. In this work, we introduce redundancy into the quantization of sensor data in order to provide several alternative representations. Notably, we trade off distortion—the difference between the sensed analog value and the digital quantization of that value—to improve lifetime. Our simulations show that when combining rate, distortion and lifetime tradeoffs we can extend Flash lifetime at a far smaller capacity cost compared to prior work. More specifically the simulated system shows that it is possible to achieve up to 2.75× less capacity cost compared to redundant Flash memory and 1.29× less capacity cost compared to the state of the art coding schemes.

CCS Concepts

• Information systems→Information storage systems
• Hardware→Robustness.

Keywords

Flash memory; Lifetime; Endurance; Embedded Storage.

1. INTRODUCTION

Embedded computer processors often record data from sensor inputs. The processor's memory logs data such as temperature, pressure, light intensity, etc., all of which are inherently analog values. The sensor thus needs to digitize the data—for example, by taking an analog temperature and converting it into a 10-bit digital value—before the processor can record it in its digital memory.

These embedded processors typically use non-volatile memory (NVM) to log these data. Today, the NVM of choice is Flash, although other technologies are emerging that may also serve this role well. Flash is dense, relatively inexpensive, and has sufficient performance (latency and bandwidth) for most applications.

The challenge with NVMs like Flash is that they wear out. A Flash cell can only be erased so many times before it can no longer be re-written. For embedded processors that cannot (easily) be accessed or repaired, this lifetime limitation can be a crucial limitation on the useful lifetime of the processor. Some applications include sensors that track driving habits in cars, animal tracking, weather balloons, and ocean probes.

The simplest method of achieving a target lifetime is to simply provide additional Flash memory, beyond the host-visible capacity, and to use that memory as portions of the Flash wear out. However, it is possible to achieve a better tradeoff between lifetime and host-visible capacity by implementing endurance codes [1, 2, 3, 4, 5, 6, 7]. The most promising schemes employ coset codes [8, 9] where data is encoded as a coset of a linear code. Each coset can be represented by multiple binary vectors (bit strings)—thus providing flexibility in what to write—and the representative is chosen to reduce the number of bits written to memory and/or promote wear-leveling. Flip-N-Write [10] is perhaps the simplest example of a coset code. The linear code is a repetition code, and every coset contains two possible representatives, a binary vector and its complement. The coset codes proposed in FlipMin [9, 11] are based on convolutional codes. More recent work on data shaping [12] combines lossless data compression and endurance coding into a single encoding operation that transforms structured data into a sequence that induces less cell wear.

All of these prior coding schemes make trade-offs between memory lifetime and *rate*. The rate of a coding scheme is defined as the ratio between the host-visible capacity and the raw capacity of the memory. As an example, Flip-N-Write [10], at the byte granularity, adds an extra bit that represents the choice of coset representative between a vector and its complement, thus creating a rate of 8/9. High rate codes are attractive because of their low cost, but substantial lifetime gains only come with lower rates. We refer to these previous schemes as rate/lifetime (RL) coding.

This paper goes back to the analog sensing application to introduce a new degree of freedom, that of *distortion*, which is the distance between the actual analog value and the quantized digital value that is recorded. The tradeoff between the number of bits used to quantize the analog signal, and the resulting distortion, is governed by rate-distortion theory [13]. However, we are not interested in finding a single quantization point that minimizes distortion, but rather in finding several alternative quantization points, each with bounded distortion. We refer to this new tradeoff as lifetime/distortion (LD) coding.

In this paper, we present the first coding schemes that combine RL and LD coding. We map each of the alternative quantization points to different cosets and thus provide a larger set of coset representatives to select from. We refer to these schemes as rate/lifetime/distortion (RLD) coding as they enable tunable tradeoffs between rate, lifetime, and distortion.

We also present a novel scheme for using higher-precision sensors to enable us to obtain the benefits of RLD techniques without suffering any increase in distortion. As a simple example, consider an embedded processor that achieves a target distortion using a 10-bit sensor. Replacing the 10-bit sensor with a 12-bit sensor can provide several alternative quantization points, each represented by a 12-bit data vector, and each able to represent the signal within the target distortion (or better). We can compare the cost of writing each 12-bit data vector to memory and select the alternative that maximizes memory lifetime. We show that this coding scheme can achieve a range of lifetime gains (up to 12×) for up to 2.75× less capacity cost compared to redundant Flash memory and 1.29× less capacity cost compared to the most promising endurance coding schemes that prior works have proposed.

Our RLD coding schemes, *like all coding schemes*, have a cost in terms of rate. In this work, we focus specifically on embedded systems where Flash durability is extremely critical and repair is impractical. High cost (i.e., low rate) coding schemes are practical for these systems, whereas they are unlikely to be attractive for typical consumer electronics or servers.

2. FLASH MEMORY

Flash memory currently dominates the market for non-volatile storage. It provides a low latency, high bandwidth, and high density alternative to hard disk drives. Because of all those benefits, Flash can be found in most modern computer systems, from enterprise datacenters to personal computers and embedded systems.

Before explaining our coding schemes, we first provide a primer on Flash memory and its characteristics that are relevant to our work. Most importantly, we must explain how Flash wears out.

2.1. Organization

Flash memory is organized in blocks, and each block contains some number of pages (e.g., 256 pages/block). Each page contains some number of Flash cells (e.g., 8192 cells/page) that are used to store bits.

Flash cells come in several varieties that are distinguished by how many levels they have. A single-level cell (SLC) has two levels[1] and can thus hold one bit (i.e., one level corresponds to 0 and the other level corresponds to 1). Flash cells can have more than two levels, and 4-level cells (MLCs) are most common. In uncoded usage, a cell with L levels can hold $\log_2 L$ bits.

2.2. Reading, Writing, and Erasing

Flash can be accessed (read or written) only at the page granularity. A Flash cell stores information by trapping charge, and charge can be added incrementally to move from one level to another. However, charge can only be removed in its entirety through erasing; erasing brings the cell's level to its lowest level.

For example, a 4-level cell (MLC) has four different levels of charge (levels L0, L1, L2, and L3), and thus it can theoretically be used to either store two bits or to store one bit that can be re-written three times (before needing to erase) as seen in Figure 1. The latter option is more attractive for lifetime gain. However, as prior work has shown [14, 8], real-life MLCs do not allow for any arbitrary transition from a lower level to a higher level. Instead due to constraints imposed by the physical layer and the Flash translation layer, 4-level cells can only be practically used to store 2 bits. Similar is the situation for cells with more levels.

To overcome this problem, prior work [8] has proposed to alter the Flash translation layer and provide an interface with "ideal cells" (or virtual cells). These ideal cells can be composed from any underlaying cell technology (SLC, MLC, etc.), but they incur a capacity cost. For example, we can take a group of Flash cells (of whatever kind) that has a capacity of C bits and use it to implement a group of $C/3$ ideal 4-level cells, where each cell now stores one bit that can be re-written 3 times. *For the rest of this work, we assume as a baseline a Flash memory with ideal 4-level cells; our work and all comparison schemes start from this same baseline.*

Although pages are the granularity for reading/writing, blocks are the granularity for erasing. Thus, all the pages of a single block must be erased at the same time. This granularity of erasing is significant because we might not want to erase all of them at the same time. For a variety of reasons, updates of data in Flash are performed "out of place". When we want to update the contents of a page we mark that page as invalid and pick a new, clean page to write our new data. Eventually we run out of clean pages and need to reclaim some previously used pages. Ideally, we could find a block that has only invalid pages on it, but often blocks have some number of valid pages on them that need to get moved before the block can be erased; the extra writes to move these valid pages is a cost called write amplification.

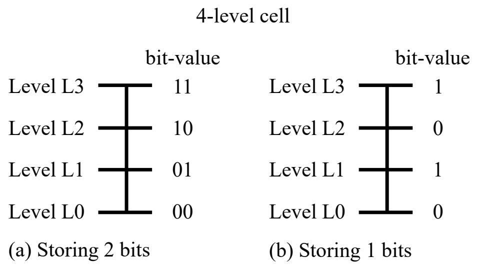

Figure 1. An ideal 4-level cell can either store two or one bits

2.3. Flash Wearout and Mitigation

Flash wears out because of its physics. During a block erasure, the trapped charge of its cells is released. This process can slowly damage the cells and cause them to degrade [14, 15]. Eventually the cells become unable to retain their charge (and thus store bits). That means that after a certain number of erases the cells are unable to be written again. The number of erases before the cells wear out depends primarily on the Flash technology and the number of levels per cell.

To improve the lifetime of Flash, one must increase the number of times cells can be written before needing to be erased. The key idea is that a Flash page can be re-written as long as every cell in that page can be re-written, i.e., no cell needs to have its charge level decreased [16]. (Recall that decreasing the charge level can only be achieved by erasing a whole block.)

Figure 2(a) presents an example of a valid Flash page re-write. Each square represents a 4-level cell of the page. The number inside the square represents the level of the cell (L0-L3) and it corresponds to a bit value. Observe that we only re-write the cells whose corresponding bits need to change. Figure 2(b) shows an example of an invalid update. The update fails this time as we need to write a cell that is already saturated. To achieve that, we need to erase that cell. However, as we already discussed, erasing individual cells is not possible in Flash memory. It is clear from Figure 2 that the bit-sequences of successive updates can dictate the number of re-writes.

In this work, we carefully pick the bit sequences we write on a page in order to maximize the number of possible re-writes and thus lifetime.

3. RATE/LIFETIME (RL) CODING BACKGROUND

There is a long history of using RL coding to improve the lifetime of Flash, and our RLD codes build on this literature. In this section, we discuss the most promising RL coding scheme for Flash (coset

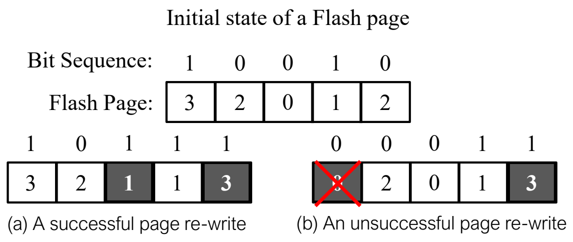
Initial state of a Flash page
Bit Sequence: 1 0 0 1 0
Flash Page: 3 2 0 1 2
1 0 1 1 1
3 2 1 1 3
0 0 0 1 1
3 2 0 1 3
(a) A successful page re-write
(b) An unsuccessful page re-write

Figure 2. Re-writing a simplistically short page with five 4-level cells (i.e., each cell can have value 0, 1, 2, or 3).

[1] The name SLC is standard, even though it is perhaps counter-intuitive that a single-level cell has more than a single level.

codes), and we discuss a more traditional error correction code (Reed-Solomon codes) that we use in conjunction with coset coding.

3.1. Coset Codes

Coset codes were introduced by Forney [17, 18] and they have been used before as RL codes for Flash [8, 9]. They show the greatest promise for Flash, and we also incorporate them into our RLD codes. We now present a brief primer on coset codes.

3.1.1. Using Coset Codes

A conventional error detection/correction code (e.g., Hamming) takes as an input a k-bit dataword and produces an n-bit codeword. The rate of the code is k/n.

A coset code is different in that it maps a k-bit dataword to a unique group of n-bit codewords called a coset. Each coset contains 2^m codewords, where $n=k+m$. Thus, each dataword can be represented by *any* of the 2^m codewords (known as coset representatives) that belong to its unique coset. This procedure is shown in Figure 3.

The benefit of coset coding—both for prior RL codes and for our new RLD codes—is that it provides multiple options for what to write. An RL coding scheme can choose which coset representative to write so as to optimize an objective (e.g., intra-word wear leveling). To select a codeword from a coset requires a metric function, and we can choose this function based on our desired goals. We assign costs to each codeword and pick the one with the least cost.

3.1.2. Generating Coset Codes

Coset codes can be generated by using a linear code C, and C can either be a convolutional code or a block code. Prior work on extending Flash lifetime [8, 9] has mainly focused on coset codes that are generated through convolutional codes that enable one to encode long datawords (i.e., thousands of bits). However, these codes are less useful for encoding sensor data where the datawords may be very short (i.e., tens of bits). For this reason, in this work, we focus on block codes that we organize in such a manner to allow for efficient coset generation as well as coset representative selection.

We demonstrate the process of coset generation through a simple example. We use the Hamming(7,4) code as C, to create a coset code with n=7, k=3, and m=4. This code has 3-bit datawords, 7-bit codewords, and 2^4 codewords per coset. Its rate is 3/7.

The generator matrix G' of the Hamming code is,

$$G' = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{bmatrix}$$

We reduce G' to its echelon form G=$[I_{n-k}|B]$,

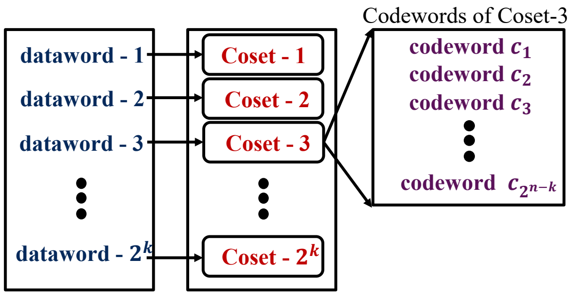

Figure 3. Mapping datawords to cosets to codewords

$$G = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix}$$

The code C contains p = 2^m codewords, each codeword of the form $c = w \times G$, for some m-bit string w.

$$C = \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ \vdots \\ c_p \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ & & & \vdots & & & \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

Now let us assume that we want to encode the dataword $\boldsymbol{a} = [101]$. Every dataword maps to a unique coset, and every coset has a unique coset label of the form $\boldsymbol{q} = [\boldsymbol{0}_{\mathbf{m}}, \boldsymbol{a}]$. For this example,

$$\boldsymbol{q} = [0 \quad 0 \quad 0 \quad 0 \quad 1 \quad 0 \quad 1]$$

We can now generate all the coset representatives $\boldsymbol{v}$ that belong in the unique coset $\boldsymbol{q}$ by performing a logical exclusive-or operation between $\boldsymbol{q}$ and C.

$$\begin{bmatrix} \boldsymbol{v}_1 \\ \boldsymbol{v}_2 \\ \boldsymbol{v}_3 \\ \vdots \\ \boldsymbol{v}_p \end{bmatrix} = \boldsymbol{q}\ XOR\ C = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ & & & \vdots & & & \\ 1 & 1 & 1 & 1 & 0 & 1 & 0 \end{bmatrix}$$

Any of the p coset representatives above can be used to represent our dataword $\boldsymbol{a}$. In order to recover our initial data $\boldsymbol{a}$ from a coset representative $\boldsymbol{v}$ we will need the parity check matrix H of the code C. This matrix is of the form $H = [H_1|H_2]$ and for our example, in its echelon form, is simply:

$$H = \begin{bmatrix} 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}$$

$$H_1 = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \end{bmatrix}, H_2 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

By solving the equation $H_2 \times \boldsymbol{a}^T = H \times \boldsymbol{v}^T$ we can recover $\boldsymbol{a}$. In other words, we can use the parity check matrix H as a "decoding matrix." This process can be performed for any block code C with dimensions *(n,m)*.

3.1.3. Encoding Example

We now present a short example of how the metric function and the coset representatives are combined to re-write a Flash page. We assume that we want to store the dataword $\boldsymbol{a} = [101]$ and our goal is to minimize bit flips. For simplicity, assume a page that consists of a single codeword.

We initially read the page we intend to rewrite. Let us assume that its content is given by,

$$\boldsymbol{v}_{\boldsymbol{old}} = [0 \quad 0 \quad 1 \quad 0 \quad 0 \quad 1 \quad 0]$$

Now we calculate a cost for each one of the coset representatives of $\boldsymbol{a}$ according to a metric function that tries to minimize bit-flips. Note that the coset representatives that map to dataword $\boldsymbol{a}$ were generated in Section 3.1.2. For the purpose of this example, we assume that the metric function adds a cost of one for each bit flip. Thus, the cost of each representative becomes,

$$\begin{bmatrix} cost_1 \\ cost_2 \\ cost_3 \\ \vdots \\ cost_p \end{bmatrix} = \begin{bmatrix} 4 \\ 2 \\ 1 \\ \vdots \\ 3 \end{bmatrix}$$

We then select the representative with the least cost; here that is $\boldsymbol{v_3}$, which has a cost of 1. We now update the Flash page with a new codeword $\boldsymbol{v_{new}}$,

$$\boldsymbol{v_{new}} = \boldsymbol{v_3} = [0 \quad 0 \quad 1 \quad 0 \quad 0 \quad 1 \quad 1]$$

In a more realistic example, our metric function will also account for other objectives (not only minimizing bit flips), like avoiding rewriting of saturated cells. Coset representatives that cannot be used to rewrite a page (because they need to rewrite saturated cells) are given an infinite cost. If all of the coset representatives that map to a dataword have infinite cost, we declare a rewrite failure. We then proceed to mark the page as invalid and pick a new clean page on which to store our data.

3.2. Reed-Solomon Codes

Our RLD codes are based on coset coding, but they also incorporate Reed-Solomon codes to enhance their benefits. Reed-Solomon (RS) codes [19] are well known for their error correction capabilities. To encode a message with a RS code, we first need to segment it into symbols of a fixed bit length. Encoding a *K*-symbol long message with RS produces a new message that is *N*-symbols long. The encoded message can now detect and correct errors at a symbol granularity. More specifically, we can correct up to *t=N-K* erroneous symbols and detect up to 2*t*.

Figure 4 gives a naïve example of a RS code operating on a 5-symbol message where each symbol is 3-bits long. Note that the cost of this code is given by the rate *K/N*.

4. UNIFORM QUANTIZATION

Sensors use analog-to-digital converters (ADCs) to represent an analog signal as a bit string that can be digitally stored and processed. This process introduces distortion and the relationship between rate and distortion has been studied extensively in the information theory literature. Here we only consider Pulse Code Modulation (PCM), where a *b*-bit sensor has 2^b different, uniform, scalar quantization levels, each of width *Δ* [20].

Figure 5 shows a 3-bit uniform quantizer (3-bit PCM), with $2^3 = 8$ quantization levels. Any analog value within the boundaries of a quantization level is quantized to the point at the center of that level. In other words, the analog value is rounded to a discrete digital value.

Because of this rounding process, the digital value and the analog value are not identical. The distortion is the difference between the two values and is typically measured as the squared Euclidean distance. Consider a sensor with *b* bits of resolution and a range *x* of analog values. In this situation, the maximum possible distortion is given by $d=(\Delta/2)^2=(x/2^{b+1})^2$. Going back to the example of the 3-bit sensor, if we assume that we are reading values ranging from 0 to 1, then the maximum distortion is $d=0.0625^2$. Another often used metric is the mean squared error (MSE), which describes the expected average distortion and is given by $\Delta^2/12$.

5. RLD CODING

The main contribution of this paper is the introduction of a coding theoretic framework that makes it possible for computer architects to tradeoff rate, memory lifetime, and distortion.

5.1. LD Coding

We now introduce lifetime/distortion (LD) coding, in which we trade off distortion and Flash lifetime. We first show how to increase lifetime by allowing more distortion, and we later (Section 5.3) show how to obtain the same lifetime benefits with no additional distortion by increasing the resolution of the sensor.

Suppose that a *b*-bit sensor is used to record analog data, and that the digital values are then stored in Flash memory. Suppose now that

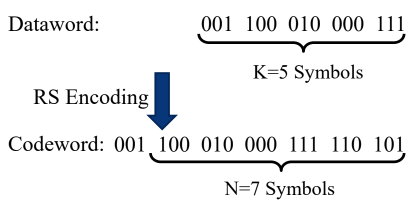
Dataword: 001 100 010 000 111
K=5 Symbols
RS Encoding
Codeword: 001 100 010 000 111 110 101
N=7 Symbols

Figure 4. Reed-Solomon example with 3-bit

an analog value, quantized as Q_L, can instead be quantized to either itself or any of its 2*M* immediate neighbors Q_{L-M}, ……., Q_L, ……., Q_{L+M}, as shown in Figure 6(a). That flexibility allows us to carefully choose the bit string to write so as to increase how many times we can re-write the Flash and thus increase its lifetime. However, distortion grows with the number of nearest neighbors *M*. The growth in distortion is predictable and *M* can be adjusted to meet specific constraints.

5.2. LD + RL = RLD Coding

To further increase the potential lifetime gains of our scheme we combine the lifetime/distortion (LD) tradeoff with a rate/lifetime (RL) tradeoff to create a RLD code. To achieve this goal, we integrate the multiple analog signal representations (quantization levels) with coset codes and Reed-Solomon codes to create RLD codes.

5.2.1. Coset Coding

We combine coset coding and LD coding in a complementary way. With LD, we take a given dataword (the quantization value) and consider multiple datawords near its value. With RL, we take a single dataword and map it to multiple codewords. The combination is RLD coding, which takes a given dataword and considers it along with nearby datawords and, for each of those candidate datawords, considers any of the codewords in its coset. If LD provides us with *2M+1* datawords and RL provides us with *C* codewords per dataword, then RLD coding provides us with *C(2M+1)* codewords from which we can choose. Intuitively, as *M* and *C* increase, we have more flexibility, but we have more distortion and lower rate, respectively.

RLD provides many options for what to write for a given dataword, and we need a metric function for choosing the best one. We borrow the metric function that was introduced by Methuselah Codes [8] to help us make that decision. This metric function is tailored specifically for Flash and helps extend its lifetime by promoting three objectives: (1) Avoid codewords that require rewriting saturated cells, (2) minimize the number of cells that need to be rewritten, and (3) balance level increments across cells.

For each one of the *2M+1* different cosets we use the metric function to select a winner codeword. From the *2M+1* winning codewords we simply use the one that has the least overall cost. We illustrate this coset selection procedure in Figure 6(b).

5.2.2. Coset Coding and Reed-Solomon

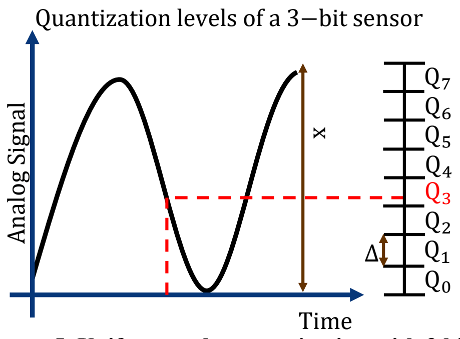

Figure 5. Uniform scalar quantization with 3 bits

The RLD code described above can provide a tunable number of options for what to write. However, sometimes even with many options, we still might not have an option we like. This situation is particularly frustrating when one or a small number of datawords with this problem cause an entire page to be un-writeable.

To mitigate this problem, we introduce a second layer of code that allows us to "skip" writing a codeword that is problematic. Rather than immediately storing each codeword after coset selection, we first buffer K of them. Codeword buffering is actually a standard practice for Flash, anyway, because we write full pages at a time (not just individual codewords).

We now use Reed-Solomon coding to skip these problematic codewords, as illustrated in Figure 6(c). We treat each of the K selected codewords as a symbol. Each symbol represents the coset of the codeword. We use Reed-Solomon coding to produce N-K additional symbols (cosets) that we write to Flash. As the new symbols represent cosets (and not codewords) we still need to perform the Viterbi algorithm in order to select the best codeword for the redundant N-K cosets. Reed-Solomon coding provides error correction at the symbol granularity, so a Reed-Solomon code that can correct t symbols would allow us to skip (not write) the t most problematic codewords (i.e., treat them as errors). That is, we could choose the t codewords with that scored the highest costs during the coset selection process. Additionally, if some codewords cannot be re-written because the Flash page is getting saturated, then we can just skip them as long as the total number of codewords that is not updated is less than or equal to t. The Reed-Solomon code can later recover the codewords that were not written.

Although this scheme can extend lifetime beyond our initial RLD code, with each additional layer of code we further decrease the code's rate because we must store more redundant bits. Thus, this Reed-Solomon coding is only optional and we can choose to use it depending on our implementation's goals and constraints.

Figure 6 presents the entire process from the moment an analog value is recorded from the sensor until it is stored into the Flash memory.

5.3. Extending Lifetime for a Fixed Distortion

We have shown that it is possible to extend lifetime at the cost of increased distortion. However, there are circumstances in which it is not acceptable to increase distortion, and so we now describe how it is possible to use a higher precision sensor to extend lifetime without increasing distortion.

To demonstrate this idea, we focus on a simple example. We assume that a b-bit sensor has been chosen for a specific implementation and the maximum allowable distortion (squared Euclidean distance) is d. Now we replace the b-bit sensor with an (b+2)-bit sensor and thus the maximum quantization error is now $d/16$. However, this time we will allow for some flexibility when recording data. When an analog value is mapped to a quantization point Q_L, we consider writing any of the three options Q_{L-1}, Q_L and Q_{L+1}. (That is, M=1). Thus, we now have three (b+2)-bit candidates to choose from when storing each sensor reading. Note that because of this process, the maximum distortion is $9d/16$ which is still lower than our initial distortion of d. We refer to this scenario as *2-bit sensor cost,* because we use a sensor with 2 extra bits of precision.

In another scenario, we replace the b-bit sensor with a (b+1)-bit sensor and again consider three quantization options per sensor recording (i.e., M=1), with maximum distortion of $9d/4$. Note that in this scenario the distortion is worse than that of the b-bit sensor but better than that of the (b-1)-bit sensor, and we show later that this design enables significant lifetime gains. We refer to this scenario as *1-bit sensor cost.*

We note that different sensor precisions lead to different MSE values. If the b-bit sensor has an MSE=$e=\Delta^2/12$, then the (b+1)-bit sensor has MSE=$e/4$, and the (b+2)-bit sensor has MSE=$e/16$.

6. IMPLEMENTATION

In this section, we explain the processes for encoding and decoding, and we discuss how we would implement these processes in hardware. Although some of these processes may appear complicated at first, most of them can be reduced to simple logic.

6.1. Encoding

The encoding process involves three steps: (1) map a dataword (quantization point) to its corresponding coset, (2) generate the coset representatives, and (3) search the coset to select the best representative according to our metric function. As we presented in Section 3.1, mapping a dataword to its coset is simply done by prepending a string of 0s to the dataword in order to get the coset label q. In other words, there is a direct mapping from quantization levels to cosets. A simple up-down counter can generate all of the quantization levels/cosets we consider for each sensor reading.

The most critical and computation challenging process are the last two steps. We present the three possible ways to generate the coset representatives and perform the search to select the best of these codewords. For reasons explained below, we implement the third option.

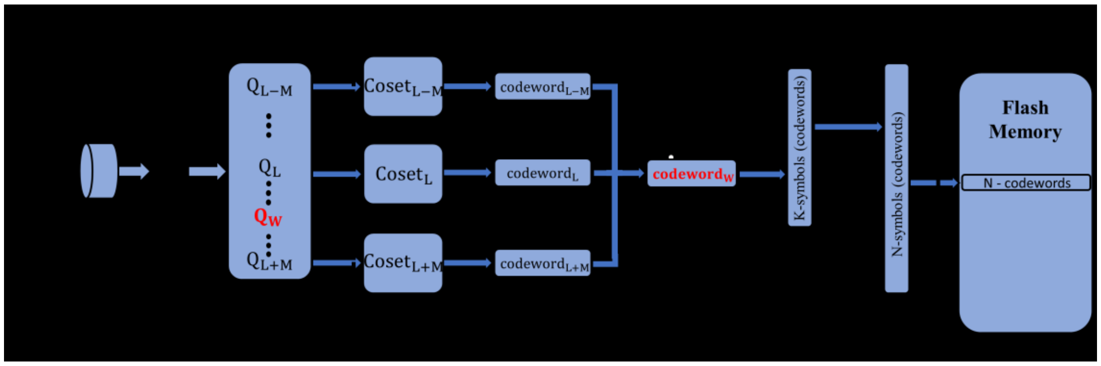

Figure 6. RLD Coding. Q_w is the quantization point corresponding to Coset_w, and Coset_w has the codeword with the least cost among all codewords in all 2M+1 cosets. With Reed-Solomon, we buffer K symbols (i.e., codewords that were produced during coset selection) and then encode them as N symbols; these N symbols are what ultimately get written to the Flash memory

The first option is to straightforwardly implement the matrix computations presented in Section 3.1.2. Generating the coset through the generator matrix of a block code requires substantial computation (matrix multiplication) but only minimal storage overhead as we would only need to store the generator matrix G and the decoding matrix D. Additionally, after generating the coset representatives, we would need to apply the metric function though a brute force approach to select the best codeword. However brute force algorithms may incur high latencies.

The second option is to replace the on-the-fly computation of the first option by generating and storing all the cosets and their representatives in look-up tables before we start recording data with our sensor. This computation only needs to be performed once but incurs additional capacity overheads that become more significant for longer codes. Additionally, we would still need to apply brute force search in order to select the desired representative.

The third option—and the one we implement—is using the Viterbi algorithm to efficiently generate and at the same time search the representatives for the one with the least cost. The Viterbi algorithm is based on dynamic programming and can efficiently generate and select the best representative. However, in order to use the Viterbi algorithm, we first need to express our code as paths through a trellis (see Forney [17, 18]). Although this is an easy process when using convolution codes (as the code *C*), there is an additional level of complexity when doing so for block codes. Prior work [17, 18, 21] has shown how block codes can be described by trellis paths by adding some constraints on the last paths of the trellis. Because of these constraints, the Viterbi algorithm needs to be executed multiple times to find the best codeword. More specifically, if the trellis has *S* states we need to run the Viterbi algorithm *S* times; these *S* executions can be parallelized. Prior work [8, 9] has also used the Viterbi algorithm to generate coset codes and select the representatives with the least cost. Viterbi incurs minimal capacity and latency overheads compared to the two previous options. We only consider codes with 4-16 states at each trellis stage and trellis states with out-degree of 2.

In Figure 7, we present all of the hardware components needed for the encoding process. Note that the codeword selection process, performed by a Viterbi hardware accelerator, is repeated serially for each quantization level. Although we could parallelize the procedure, we choose not to in order to minimize the amount of hardware and power needed. The selected codeword is stored along with its cost. A comparator later selects the codeword with the overall least cost. We also note that we need to buffer a single Flash page, because the Viterbi accelerator needs to know the current state of the Flash page in order to select the best coset representative. The final codeword is first stored in the buffered Flash page. When we are done re-writing the whole page, we proceed to store it in the actual Flash memory. The next page we intend to re-write is loaded into the buffer and the process is repeated.

If we choose to also apply the Reed-Solomon optimization, we also need an additional encoding matrix. The RS uses a simple binary vector for matrix multiplication. The cost of that multiplication can be amortized through the number of symbols we encode, as each symbol represents a recorded value from our sensor. Thus, the most critical part is the Viterbi accelerator that needs to execute multiple computations for every value that the sensor records.

In Section 8.4, we analyze the cost of the Viterbi hardware accelerator.

6.2. Decoding

To decode a codeword (i.e., a coset representative), we just need to multiply it with the parity check matrix, *H*, as discussed in Section 3.1.2, in order to recover the initial dataword. Although this may sound difficult, the parity check matrix is usually a sparse binary matrix and thus the multiplication can be reduced to several logical XOR operations.

In the case we also used Reed-Solomon coding, additional processing is necessary to recover any codewords that we intentionally decided not to write. This process requires solving a number of linear equations in order to detect the location of potentially erroneous symbols. Although this process can be relatively computational intense, the latency overheads can be amortized over the number of sensors readings (*K*) that are decoded at the same time.

The decoding logic can be located either on the embedded processor or on the system that receives and consumes the embedded processor's data.

7. EXPERIMENTAL METHODOLOGY

We envision RLD coding being used primarily on embedded processors with demanding requirements for Flash endurance. These processors will have a range of tolerances for other metrics that depend on the processor's usage model, and these metrics include Flash capacity, performance, energy, power, and distortion. Because we cannot possibly evaluate all possible sensor applications, we focus mainly on one example application.

7.1. System Model

We use Matlab to simulate the behavior of our Flash memory system, both with and without our coding schemes. We simulate a Flash memory that consists of pages of *ideal 4-level cells* (recall from Section 2), and all schemes evaluated use these ideal cells. Unless otherwise specified, the baseline's sensor has 10 bits of precision.

For RLD coding, we vary the following parameters: sensor precision, amount of distortion (*M*), coset code, and Reed-Solomon rate. Nevertheless, we focus most of our experiments on a 12-bit sensor, *M*=1, and a coset code generated through a Golay(24,12) code [22]. When used, the default Reed-Solomon rate is 0.5. With these parameters, for each sensor reading, we can now select among three quantization points (cosets) where each one contains 2^{12} different 24-bit codewords.

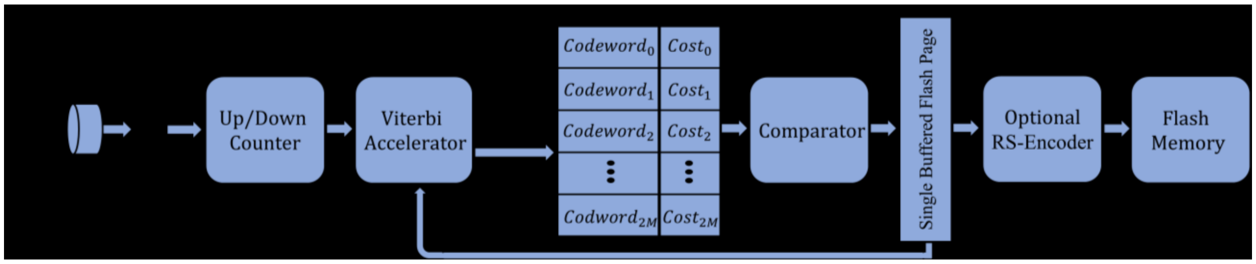

Figure 7. A Hardware Implementation of the Encoding Process

We refer to RLD codes with the following notation: RLD[coset code, value of M]. For example, when we use the Golay(24,12) coset code with M=1, we denote it as RLD[Golay(24,12), 1]. If we also use the Reed-Solomon option with a rate of s, we denote it with "+RS(s)" appended to the coset code, as in RLD[Golay(24,12)+RS(s), 1].

7.2. Experiments

To test our RLD codes and calculate their gains, we consider both random data inputs and data inputs from a sensor that records gradually increasing/decreasing light intensity as a voltage between 0 and 3.3 volts every 20 milliseconds. In the case of real data, we first record the data using a 12-bit sensor and then input them to Matlab to perform our simulation. In the case of random input data, we generate random binary vectors of length b+2 or b+1. We vary the value of b to test different codes with different rates. The simulator reads the data in the same order that were generated and forwards them to our RLD scheme to be processed.

After the data are processed they are stored to a Flash page. We simulate a single page of Flash, starting with a blank page, and re-writing data to the page, without erasure, as many times as allowable. When the cells are saturated and re-writing is no longer possible, we erase the page and start again.

Simulating a single page suffices for random data, for two reasons. First, the Flash interface performs writes "out of place" and thus effectively randomizes the data that are written each time (i.e., there is no correlation between what is being written to a page and what was previously written to it), as explained in Section 2. Second, because we run our simulations multiple times and take average results, we expect to have statistically the same results as with simulating many pages.

Simulating a single page with real data inputs is actually *less* realistic than using random data, because there will be unlikely correlations between what is being written and what was written previously. We perform this experiment to determine if these correlations greatly help our coding schemes. If so, that result would argue for "in-place" updates in Flash for sensor devices.

We measure Flash lifetime as the average number of re-writes per erase. There are two other metrics we considered, but neither is as insightful. One option is the lifetime of an entire Flash memory (or even just a page of Flash memory), but those results are proportional to our metric multiplied by a very large and not exactly known number, which is the expected number of erases a cell can tolerate before wearout. Another option for a metric is the percentage of saturated cells in a page, but this is also less informative. Consider two example scenarios: one where we re-write a page 10 times before erasing and only half the cells are saturated, and one where we re-write 2 times with all the cells being saturated. The first example is still better in terms of lifetime as we utilized the page more before erasing it.

We compare our scheme against a baseline (denoted as "Baseline") that uses each ideal 4-level cell to hold one bit and can thus write the cell 3 times before each erase, as discussed in Section 2 (i.e., its lifetime is 3) and illustrated in Figure 1(b). We also compare against an RL code (denoted as "RL") that uses the same coset code as we do.

7.3. Metrics

The primary coding metrics we consider are lifetime, distortion, and rate. We express rate as the ratio of host-visible capacity to total (raw) capacity.

One other insightful coding metric is aggregate gain, which was introduced by Mappouras et al. [8]. Aggregate gain is the product of rate and lifetime. Thus, for a given lifetime, greater aggregate gains indicate more efficient implementation, in terms of capacity cost. Consider a baseline system with no coding (i.e., rate = 1), that uses ideal 4-level cells (and thus has a lifetime of 3); its aggregate gain is equal to 3. We normalize all aggregate gains to this baseline system. Although higher aggregate gains are generally preferable, sometimes the lifetime gain may be an overkill. For example, a lifetime gain of 12× for a Flash memory may exceed the lifetime of the sensor itself and be wasteful. Thus, one may prefer an implementation with lower lifetime gains but higher rate (i.e., host-visible capacity) even if the overall aggregate gain is reduced.

Lastly, we consider implementation metrics including latency and power.

8. EVALUATION

In this section, we evaluate and compare RLD coding to the Baseline and to RL coding.

8.1. Lifetime Gain

The most important metric in this work is lifetime gain. Figure 8 presents the lifetime gains for the Baseline, RL[Golay(24,12)], and RLD[Golay(24,12), 1]. Recall that the Baseline achieves 3 writes per erase.

Our simulations show that, on random data, RLD achieves a lifetime of 14.8 writes per erase, which is 15.6% better than RL's 12.8 writes per erase. We see that RLD provides an even greater advantage when considering real data. The reason behind that is because the recorded values often change slowly over time and thus consecutive updates exhibit high correlation. This characteristic—which may not occur in typical Flash—allows our distortion/lifetime tradeoff to maximize its gains and demonstrate that "in-place" updates may be a more suitable interface for sensor devices.

In Figure 9, we present the lifetime of our RLD+RS code, as a function of its rate s. We fix the coset code (Golay(24,12)) and the distortion (M=1). We can see that the Reed-Solomon option can help RLD achieve remarkable lifetime gains that increase as s decreases. However there is not much difference between real and random data any more. Although we do not have any formal way to prove why this is the case, it could be that the page is already pushed to its limit of available re-writes for the given rate and distortion tradeoffs.

8.2. Capacity Loss and Aggregate Gain (AG)

Although RLD's lifetime gains seem quite promising, we have to justify the rate (host-visible capacity) loss of our scheme. For that reason, we present the aggregate gain of each code. We normalize the host-visible capacity of each code to that of the Baseline, which we denote as C. We explore the two different sensor resolution scenarios that we discussed in Section 5.3 (1-bit and 2-bit sensor costs).

In Figure 10, we graphically present the results for the 2-bit sensor cost scenario; the Baseline has a 10-bit resolution while RLD has a 12-bit resolution. We observe that, based on our simulations,

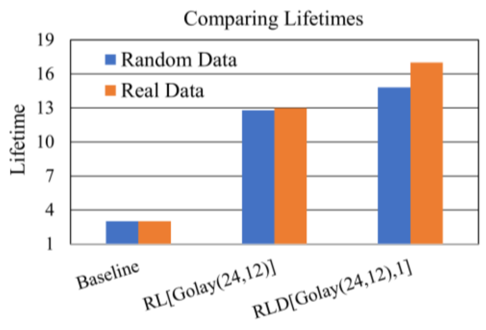

Figure 8. Comparing Lifetimes

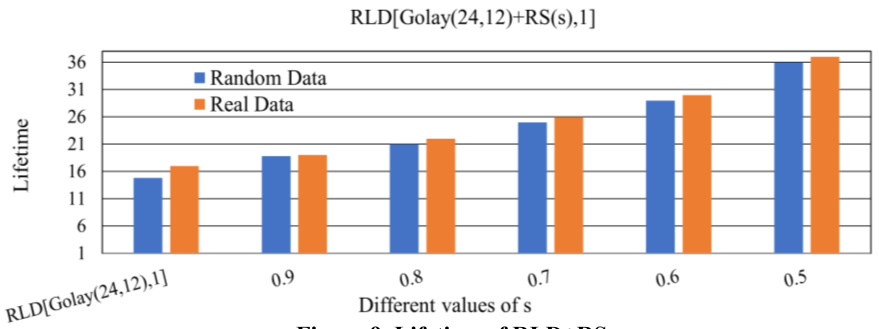

Figure 9. Lifetime of RLD+RS

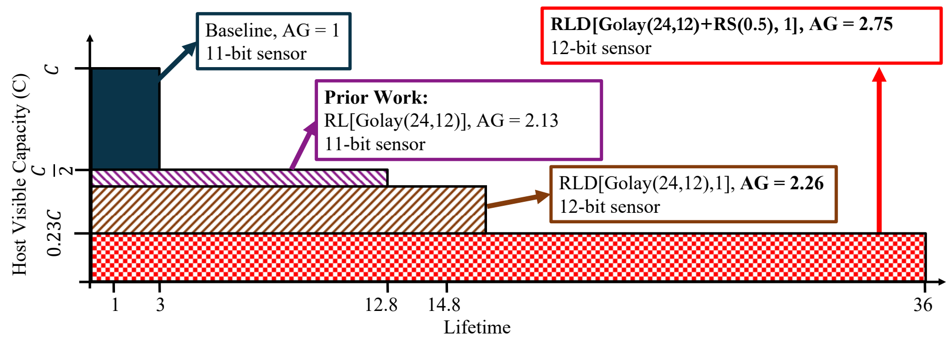

Figure 10. Aggregate gain comparison with a 10-bit sensor baseline (2-bit sensor cost scenario)

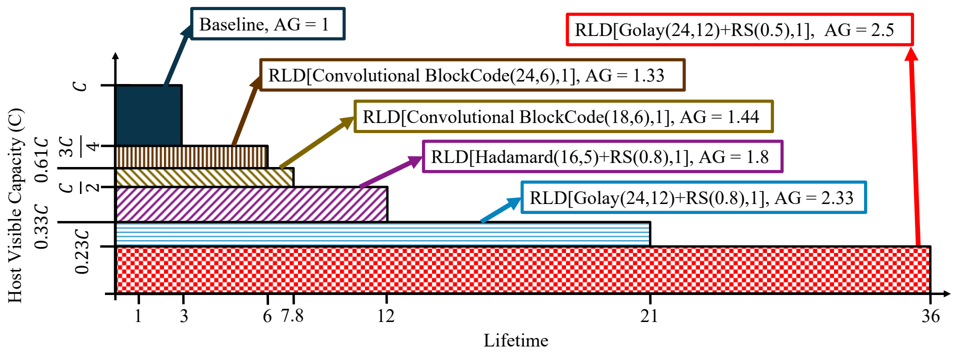

Figure 11. Aggregate gains for various coset codes.

RLD[Golay(24,12)+RS(0.5),1] achieves the best aggregate gain of 2.5 with a lifetime of 36 . That means that RLD[Golay(24,12)+RS(0.5),1] can achieve a lifetime of 36 for 2.5× less capacity cost compared to Baseline. RLD[Golay(24,12),1] achieves better lifetime than the RL code, but its aggregate gain is just less. This discrepancy is because the RLD code's host-visible capacity is 10/12 that of the RL code, in order to account for the two additional bits of its higher precision sensor.

To calculate the aggregate gains for the 1-bit sensor cost scenario we project these results for a Baseline that uses an 11-bit resolution. In this situations where some distortion is tolerable—perhaps because b-1 bits is too little resolution and b bits is more than necessary— RLD[Golay(24,12)+RS(0.5),1] achieves an aggregate gain of 2.75, while RLD[Golay(24,12),1] achieves an aggregate gain of 2.26, and thus both exceed the aggregate gain of prior work. The increase in aggregate gain is due to using only one extra bit of resolution (instead of 2).

8.3. Exploring Other RLD Codes

RLD comprises a range of codes. One can obtain the desired Flash lifetime by choosing the allowable distortion (determined by M) and rate (determined by the choice of coset code).

8.3.1. Varying the Coset Code

In Figure 11, we present the results for a fixed distortion (M=1), but a range of different coset codes. We consider coset codes that are generated from various typical block codes (Hadamard, Golay) as well as block codes that are based on convolutional codes. We also include RLD[Golay(24,12)+RS(0.5), 1], which was in the previous figure, because it is our best code in terms of aggregate gain. Comparisons are somewhat tricky in that the different coset codes have different dataword sizes. We assume the Baseline has b-bit resolution (the resolution of the Baseline does not affect its aggregate gain) and each RLD code has (b+2)-bit resolution. This added resolution introduces a cost (in terms of rate) for the RLD codes.

Our simulations show that different RLD codes can achieve better rate (greater host-visible capacity) than RLD[Golay(24,12), 1], but the aggregate gain is worse. This result is because codes that have a rate close to 0.5, like Golay(24,12), have been shown to "behave better". However, no formal proof exists, in the field of information theory, that can explain this common observation between codes of different rate.

8.3.2. Varying the Distortion

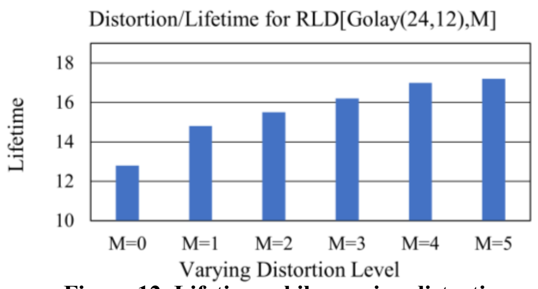

Figure 12. Lifetime while varying distortion

We now consider the impact of distortion beyond *M*=1. In Figure 12, we present results for RLD[Golay(24,12),*M*] as we vary the amount of distortion. When *M*=0, this code is the same as RL[Golay(24,12)].

We observe that as we increase the value of *M*, lifetime initially increases linearly. However, beyond *M*=4, we see a diminishing lifetime gain. The reason behind that is that we reach a point where adding more quantization points does not provide significant variation during the codeword selection process. To further explain this effect, one can imagine that we are trying to re-write an initially erased *p*-bit page. There are many codewords (*p* to be precise) that can re-write the page with only flipping a single bit. However, all of them are equivalent in terms of lifetime gain and thus adding more options does not result in significant lifetime gains.

Further increasing *M* will eventually result in extremely high lifetime gains (until we reach infinite lifetime) because we will rarely have to re-write our memory as almost every quantization point will be considered for every recorded value. However, we are not interested in such a scenario as it results in no actual information stored to our memory because distortion also reaches infinity.

8.4. Power and Latency

Any RLD (or RL) implementation must incur latency and power overheads, particularly because of the coset generation and selection process. As we already discussed, we can use a Viterbi accelerator to perform this process. For the RLD[Golay(24,12), M] implementation we use a single hardware accelerator. Because the RLD[Golay(24,12), M] implementation is based on a 16-state block code (i.e., Golay(24,12)), the accelerator needs to run 16 sequential times for each recorded value (see Section 6.1). Additionally, we need to repeat the whole process for every one of the three quantization points we consider. We could parallelize this process by adding several accelerator devices, if needed. We note that the latency of this process must never exceed the sampling rate of our sensor, otherwise values will be recorded faster than they can be encoded.

There are several proposed Viterbi accelerators [23, 24, 25] that can achieve low latency and power. Using the power and latency overheads that are reported in those papers we can predict the overheads of our implementation. It is worth mentioning that these accelerators target implementations that use lengthy inputs (~100s of bits). Our implementation, however, only requires 12-bit inputs. Thus, the latency and power we report are on the high end of what should be expected.

The Viterbi accelerator reported by Azhar et al. [25] has a latency of ~70ns. Because we need to run 16×3=48 sequential times, our latency is 3.36ms, which is well below the 20ms sampling rate of our sensor. The power of the accelerator is just 12.4mW and it runs at 370MHz. Other Viterbi accelerators have lower latencies but higher power overheads. However, for our implementation, we prefer lower power as the latency is already sufficiently low.

Other costs of our implementation include the intermediate buffers shown in Figure 7. The amount of buffering needed depends on the details of the implementation. For the RLD[Golay(24,12), M=1] this is just three 24-bit buffers and thus their area and energy overheads are not significant. Additionally, we perform an extra read before each write. This read is performed in the background, while recording data, and its latency is completely hidden. As the power of reads is about half that of writes [14], our scheme increases the power of Flash writes by 50%. However, this is an overhead that (to the best of our knowledge) all re-writing codes have to incur.

As we can see, the overheads are minimal. We should mention, however, that for different sensors that have higher sampling rates we may need to use multiple accelerators to meet the latency constrains. That will result in higher power overheads.

9. RELATED WORK

Because of its importance, there is plenty of prior work that tries to extend the lifetime of Flash memory. The research in this topic can be categorized into three orthogonal approaches. The first approach is using codes that enable Flash pages to be re-written in order to increase how many times a page is written before it has to be erased. A number of prior papers [1, 2, 3, 4, 5, 6, 7, 26, 27] have focused on write-once memory (WOM) codes [28] (or variations of WOM codes) to extend the lifetime of Flash memory and other non-volatile memories. However, feasible WOM codes only allow 2-4 updates while more complicated implementations become infeasible due to their high complexity. Other re-writing techniques also exist [29] that use invalid pages to re-write data but only provide minimal lifetime gains. The most promising re-writing codes presented by prior work [8, 9] were based on coset codes (similarly to our RL implementation). Because of the high flexibility of coset codes, they can achieve high lifetime gains while providing a range of solutions and tradeoffs. However, none of the prior work on re-writing codes has considered the lifetime/distortion tradeoff that we present in this work.

The second approach to extend Flash lifetime is by rethinking the organization and page management of a Flash device in order to perform wear-leveling and reduce the number of erases. Researchers have observed that the lifetime of Flash memory can be extended if we ensure that all of the blocks (and thus pages) are equally erased across the lifetime of the memory. By doing so we can assure that the memory will wear out evenly and thus prolong its overall lifetime. Prior work [30, 31, 32, 33, 34] focused on performing wear-leveling across blocks and pages by enhancing the Flash translation layer (FTL), software that is responsible for performing various operations like garbage collection. Other researchers [35] focused on the system file level in order to reduce the number of unnecessary updates to the Flash.

The third way to increase Flash lifetime is by tolerating errors that occur due to degraded cells and thus prolonging its effective time-span. Dolacek et al. [36] and Sala et al. [37] have introduced channel coding techniques to minimize bit flip errors due to degraded cells. This body of work translates physical processes to channel models and then develops channel coding strategies that remediate errors in digital data. Guo et al. [38] provide error codes for Flash memory with flexible rates. They introduce different error protection levels for compressed data that have different impact on the final signal to noise ratio. Thus, they try to maximize the tradeoff between rate and noise on compressed data on Flash memory.

We should also note that our work is relevant to approximate computing. Approximate computing [39] can be used to reduce energy and capacity as well as minimize the effect of early failures in NVMs. In this work though, we choose not to trade data accuracy and we extend lifetime beyond the point of what approximate computing can achieve.

10. CONCLUSION

Increasing the lifetime of Flash is important, particularly for embedded processors. For embedded processors that record analog data from sensors, there is an exciting opportunity to consider multiple quantization points so as to provide flexibility in what to write. We synergistically combine this flexibility in quantization with the flexibility provided by coset coding, and the resulting RLD codes provide a way to increase Flash lifetime while tuning the trade-offs with distortion and rate.

11. REFERENCES

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

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

[3] A. Jiang, V. Bohossian and J. Bruck, 2007. Floating Codes for Joint Information Storage in Write Asymmetric Memories. Int'l Symposium on Information Theory, pp.1166-1170.

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

[5] B. Kurkoski, 2013. Rewriting Flash Memories and Dirty-paper Coding. IEEE Int'l Conference on Communications, pp.4353-4357.

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

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

[8] G. Mappouras, A. Vahid, R. Calderbank and D. J. Sorin, 2016. Methuselah Flash: Rewriting Codes for Extra Long Storage Lifetime. In Int'l Conference on Dependable Systems and Networks (DSN), pp. 180-191.

[9] A. Jacobvitz, R. Calderbank and D. Sorin, 2012. 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.

[10] S. Cho and H. Lee, 2009. Flip-N-Write: A Simple Deterministic Technique to Improve PRAM Write Performance, Energy and Endurance. In Int'l Symposium on Microarchitecture, pp. 347-357.

[11] A. Jacobvitz, R. Calderbank and D. J. Sorin, 2013. Coset Coding to Extend The Lifetime of Memory. In Int'l Symposium on High Performance Computer Architecture (HPCA), pp. 23-27.

[12] Y. Liu and P. H. Siegel, 2017. Shaping Codes for Structured Data. In Proc. of the IEEE Global Communications Conference..

[13] T. Berger, 1971. Rate Distortion Theory: A Mathematical Basis for Data Compression. Prentice-Hall, Englewood Cliffs.

[14] L. M. Grupp, A. M. Caulfield, J. Coburn, S. Swanson, E. Yaakobi, P. H. Siegel and J. K. Wolf, 2009. Characterizing Flash Memory: Anomalies, Observations, and Applications. In 42nd Annual Int'l Symposium on Microarchitecture, pp. 24-33.

[15] V. Mohan, T. Siddiqua, S. Gurumurthi, and M. R. Stan, 2010. 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.

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

[17] G. J. Forney, 1988. Coset Codes. I. Introduction and Geometrical Classification. Trans. on Information Theory, vol.34, no.5, pp.1123-1151.

[18] G. J. Forney, 1988. Coset Codes. II. Binary Lattices and Related Codes. Trans. on Information Theory, vol.34, no.5, pp.1152-1187.

[19] I. S. Reed and G. Solomon, 1960. Polynomial codes over certain finite fields. Journal of The Society For Industrial and Applied Mathematics, vol. 8, pp.300-304.

[20] N. S. Jayant and P. Noll, 1984. Digital Coding of Waveforms: Priciples and Application to Speech and Video. Englewood Cliffs, pp.115-251.

[21] R. Calderbank, G. D. Forney and A. Vardy, 1999. Minimal Tail-Biting Trellises: The Golay Code and More Trans. on Information Theory, vol. 45, no. 5, pp. 1435-1455.

[22] M. J. Golay, 1949. Notes on digital coding. Proc. of The Institute of Radio Engineers, vol. 37, no. 6, pp. 657-657.

[23] C. Cheng and K. K. Parhi, 2008. Hardware Efficient Low-Latency Architecture for High Throughput Rate Viterbi Decoders. Trans. on Circuits and Systems II: vol. 55, no. 12, pp. 1254-1258.

[24] A. R. Buzdar, L. Sun, M. W. Azhar, M. I. Khan and R. Kashif, 2017. Area and Energy Efficient Viterbi Accelerator for Embedded Processor Datapaths. Int'l Journal of Advanced Computer Science And Applications, 402-407, 2017.

[25] M. W. Azhar, M. Själander, H. Ali, A. Vijayashekar, T. T. Hoang, K. K. Ansari and P. Larsson-Edefors, 2012. Viterbi Accelerator for Embedded Processor Datapaths. In 23rd Int'l Conference on Application-Specific Systems, Architectures and Processors (ASAP), pp. 133-140.

[26] A. Eslami, A. Velasco, A. Vahid, G. Mappouras, R. Calderbank and D. J. Sorin, 2015. Writing without Disturb on Phase Change Memories by Integrating Coding and Layout Design. In Proc. of the 2015 Int'l Symposium on Memory Systems, pp.71-77.

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

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

[29] G. Yadgar, E. Yaakobi and A. Schuster, 2015. Write Once, Get 50% Free: Saving SSD Erase Costs Using WOM Codes. in FAST, pp. 257-271.

[30] L.-P. Chang, 2007. On Efficient Wear Leveling for Large-scale Flash-memory Storage Systems. In Proc. of the 2007 ACM Symposium on Applied Computing (SAC), pp.1126-1130.

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

[32] L.-P. Chang and C.-D. Du, 2019. Design and Implementation of an Efficient Wear-leveling Algorithm for Solid-state-disk Microcontrollers. ACM Trans. on Design Automation of Electronic Systems (TODAES), vol. 15, no. 1, pp.1-36.

[33] M. Murugan and D. H.-C. Du, 2011. Rejuvenator: A Static Wear Leveling Algorithm for NAND Flash Memory With Minimized Overhead. In 27th Symposium on Mass Storage Systems and Technologies (MSST), pp. 1-12.

[34] X. Jimenez, D. Novo and P. Ienne, 2014. Wear Unleveling: Improving NAND Flash Lifetime by Balancing Page Endurance. In FAST, vol. 14, pp. 47-59.

[35] Y. Lu, J. Shu and W. Zheng, 2013. Extending the Lifetime of Flash-Based Storage Through Reducing Write Amplification From File Systems. In FAST, vol. 13.

[36] L. Dolecek and Y. Cassuto, 2017. Channel coding for non-volatile memory technologies: Theoretical advances and practical considerations. In Proc. of the IEEE, vol 105, pp. 1705-1724.

[37] F. Sala, K. A. S. Immink, and L. Dolecek, 2015. Error control schemes for modern flash memories: Solutions for Flash deficiencies. IEEE Consumer Electronics, vol. 4, pp. 66-73.

[38] Q. Guo, K. Strauss, L. Ceze and H. S. Malvar, 2016. High-density image storage using approximate memory cells. In ACM SIGPLAN Notices, vol. 51, no. 4, pp. 413-426.

[39] S. Mittal, 2016. A survey of techniques for approximate computing. ACM Computing Surveys (CSUR) vol. 48, no.4, p.62.