Writing Cosets of a Convolutional Code to Increase the Lifetime of Flash Memory

by

Amay Jhaveri

Advisor: Dr. Dan Sorin, Professor of Electrical and Computer Engineering

Duke University

Department of Electrical and Computer Engineering

2014

Table of Contents

1 Abstract

There has recently been a trend in the use of solid-state drives (SSDs) for mass storage. These solid-state drives are made from flash memory cells and have numerous advantages over hard disk drives. While SSDs are very attractive to use, the flash memory cells that make up an SSD are prone to wear out; there is a limit to the number of times a cell can be erased and re-written and this limit is meant to reduce in the future [3]. In order to address this, Jacobvitz et al. [1] proposed an endurance-coding scheme that delays the erasure of a flash page. We extended his work by applying a rate ¼ convolutional code for encoding bits being written to an SSD and observed the write efficiency by doing so. The rate ¼ convolutional code had a low area overhead of 35%, but this came at the tradeoff of a 100% improvement in write efficiency for 4-level cells and no improvement in write efficiency for 2-level cells. We also applied different metrics used in Viterbi decoding for encoding bits, but this resulted in no further improvements in write efficiency. The metrics however, did show us the importance of bit-flip reduction in obtaining higher write efficiencies.

2 Introduction

The trend in mass storage is moving towards solid-state drives over hard disk drives. This is primarily because SSDs have many advantages such as better performance, higher access speeds and lower energy consumption. While SSDs may be more expensive, the cost is slowly decreasing over time making SSDs more affordable to be used in personal computing as well as datacenters [2].

SSDs are made up of flash memory cells. According to Moore's law [8] we should be able to increase storage over time due to the packing of more cells, but this however, leads to issues with robustness. The long-standing issue with flash memory cells is they are prone to wear-out i.e. they can only be erased approximately 10^4 times before they need to be replaced [3]. Therefore, while storage space may be increasing, the SSDs may no longer be as durable and hence be rendered useless.

In order to address the problem, Jacobvitz et al [1] apply endurance coding techniques to write multiple times to a flash page before erasing it. This research has shown that endurance coding techniques can indeed increase the lifetime of SSDs over 500%. In his work, Jacobvitz shows the application of a rate ½ convolutional code to allow multiple writes to a flash page before its erasure; he notes however, that using such a code requires a 100% area overhead, which is greatly undesirable. To reduce the area overhead, we show how a rate ¼ convolutional code can be applied to a flash page while still allowing multiple writes to the page before erasure.

In this work, we first look at efficiently applying a rate ¼ convolutional code to a Flash SSD and compare it to the results from a rate ½ convolutional code. We then dive deeper into modifying the Viterbi decoding algorithm used with the rate ¼ code to improve wear leveling. This modification primarily involves changing the metric that the algorithm uses to choose the best path for encoding data. We apply a variety of modified metrics along with their results for number of writes to a flash page before erasure.

3 Overview

3.1 Solid State Drives

SSDs are made up of flash memory cells. Each flash memory cell is made up of multiple levels; these levels can store different states of charge before the flash memory cell reaches saturation. Commonly used flash cells are 2-level cells (2LCs), 4LCs, 8LCs. The level of a flash cell can only increase, but once a cell reaches saturation (a 4-level cell reaches saturation at level 3), it must reset to level 0. Using a technique known as waterfall coding [10], we use each level to only hold 1 bit of data. A flash cell can only be reset by an erasure, and there are a limited number of erases before a flash cell wears out.

Flash SSDs are organized in pages, and pages are grouped in blocks. Data is written to SSDs at the page granularity, whereas SSDs are erased at the block granularity [11]. Once a single flash memory cell in a page in the block reaches saturation, the next write to that cell requires an erase of the entire block; this means pages holding valid data within the block need to be copied over to a new empty block. We can quickly see why it is important to erase a page as late as possible and have as many pages close to saturation before erasing. The copying of data from one set of pages in one block to another set in a new block is known as write-amplification [4], which drastically increases the wear on a flash SSD.

3.2 Convolutional Codes

A convolutional code is a type of error correcting code where information that requires m bits of storage is encoded such that it then requires n bits of storage. m/n is known as the rate. The function to transform the information requires the last k pieces of information, where k is the constraint length of the convolutional code. The transformation function is represented by generator polynomials, where the number of generator polynomials is equal to n and the number of operands in each polynomial is equal to k.

For example, a rate $^1/_3$ convolutional code with a constraint length of 3 will have generator polynomials as shown in Equations 3.1.

$$\begin{aligned} g_0 &= 1,1,1 \\ g_1 &= 0,1,1 \\ g_2 &= 1,0,1 \end{aligned} \tag{3.1}$$

The final output bits using the generator polynomials are show in Equations 3.2. The final result is modulo-2 since it should be in binary.

$$\begin{aligned} n_0 &= m_1 + m_0 + m_{-1} \\ n_1 &= m_0 + m_{-1} \\ n_2 &= m_1 + m_{-1} \end{aligned} \tag{3.2}$$

A block diagram view of the convolutional encoder with shift registers can be seen in Figure 3.3.

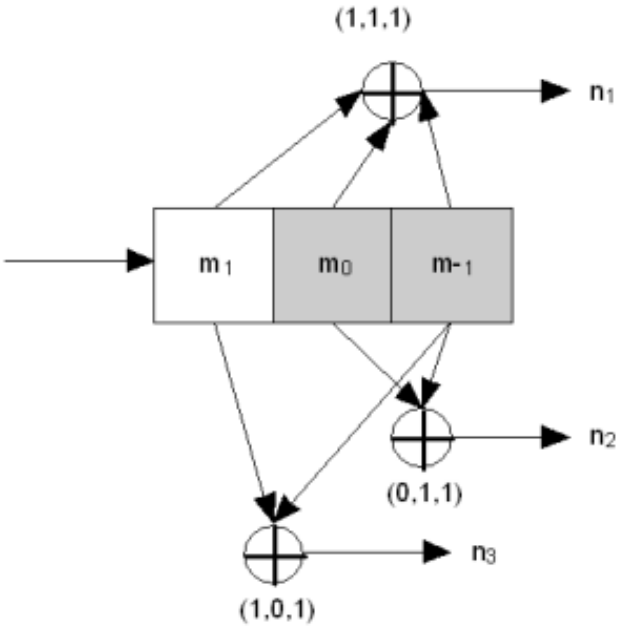

Figure 3.3: Block Diagram for a rate $^1/_3$ convolutional encoder with $k = 3$

The $\oplus$ symbol is the XOR symbol and works as a modulo-2 adder.

The convolutional code can be implemented using a state machine where the number of states is 2^{k-1}. The system only processes one bit at a time, and this bit determines the next state. The current state is determined by examining the contents of the left most k-1 shift registers at a given time i.e. m_1 and m_0 in Figure 3.3.

The state machine described above can be represented using two tables – the next state table (Table 3.4) and the output table (Table 3.5). For a rate $^1/_n$ convolutional code, the number of bits in the output is equal to n.

Table 3.4: Example of Next State Table for $k = 3$

Current State	**Input = 0**	**Input = 1**
00	00	10
01	00	10
10	01	11
11	01	11

Table 3.5: Output Table for a Rate ½ code and $k = 3$

Current State	Input = 0	Input = 1
00	00	11
01	11	00
10	10	01
11	01	10

3.3 The Trellis Diagram and Viterbi Decoding

A trellis is a graph whose nodes are organized in vertical slices (generally time-steps) where nodes in one slice are connected to at least one node in prior and subsequent slices. Using Table 3.4 and Table 3.5, a trellis diagram is constructed assuming 00 as a start state. An example of a trellis diagram using the above tables can be seen in Figure 3.6.

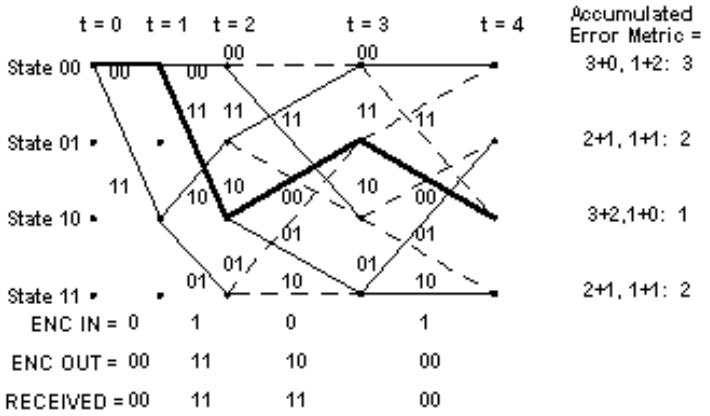

Figure 3.6: Trellis Diagram for 4 time-steps

The trellis computes all possible paths (cosets of the convolutional code) based on the input bit and the current state. The Viterbi algorithm [6] then uses this trellis to calculate error metrics for each state at each time-step based on the received bits and the output bits at each state. We will refer to the output bits at each state as an edge. In the case of Figure 3.6, the error metric is just the Hamming distance [7] between the received bits and the encoded bits at each edge. The final path that is chosen is the one with the lowest cumulative error metric at the last possible time-step. Based on the chosen path, the encoded bits are then calculated.

3.4 Encoding the Data Bits during a Write

Encoding an n-bit sequence (referred to as a dataword) involves a three-step process [1].

1. The data word is converted to a codeword by multiplying it with an Error Correcting Code (ECC) matrix generated based on the convolutional code chosen. The convolutional codes are chosen such that they have maximally distant properties [5]. Since the convolutional code is also used to create the trellis, it is compatible with the ECC codeword.

2. XORing the codeword currently stored in the SSD with the ECC codeword generated in Step 1. This helps identify the number of extra bits to be written based on the difference between the two codewords.

3. Using the Viterbi algorithm to explore the set of all possible codewords and then finding the most optimal (closely matching) codeword based on the result from Step 2. Thus, we see that the Viterbi decoding algorithm is being used to encode. Looking at Figure 3.6, the "Received" bits are analogous to the codeword from Step 2, and the final encoded word written to the SSD is analogous to the "Enc Out" bits.

4 Implementation

4.1 The Error Correcting Code (ECC) Matrices

Building upon the work done by Jacobvitz et al., a MATLAB script was written to generate a $^1/_n$ feed forward matrix for the first step of the encoding process. A generator matrix with rate $^1/_n$, constraint length k, and generator polynomials $g_0, g_1, \ldots, g_{n-1}$ will take the following form:

$$G = \begin{pmatrix} G_0 & G_1 & G_2 & \cdots & G_{k-1} & 0 & 0 \\ 0 & G_0 & G_1 & G_2 & \cdots & G_{k-1} & 0 \\ 0 & 0 & G_0 & G_1 & G_2 & \cdots & G_{k-1} \\ 0 & 0 & 0 & \ddots & \cdots & \cdots & \cdots \end{pmatrix}$$

where each $G_i = \begin{pmatrix} g_0^{(i)} & g_1^{(i)} & \cdots & g_{n-1}^{(i)} \end{pmatrix}$, and i represents the i^{th} bit in a generator polynomial.

The number of rows depends on the size of the dataword to be encoded since this generator matrix is to be multiplied by the dataword to get a codeword. The number of columns is the size of the codeword. The row wraps around if G_0 to G_{k-1} reaches the last column of the matrix. All information to construct the codes, primarily the generator polynomials that have maximal distance properties, can be found in in Table 12.1 (c) of Lin and Costello's textbook [5].

In order to calculate the dataword length given the codeword length, we consider the following:

- Storage overhead required for error correction: $log_2(codeword\ length) + 1$
- History Length based on rate of convolutional code: $(codeword\ length)/n$

The dataword length can then be calculated using Equation 4.1.

$$dataword\ length = (codeword\ length) - (history\ length) - (error\ correction\ overhead) \quad (4.1)$$

In our work, we primarily looked at rate ¼ codes and compared them to the rate ½ codes that were used by Jacobvitz et al. Once the MATLAB script for generating matrices was complete, we ran the following regression tests to verify the correctness of the matrices:

- Ensuring that the generator polynomials were successfully combined to create an interleaved polynomial i.e. $G_0 \ldots G_{k-1}$.
- Verifying that the rank of the generated matrix over GF(2) equals the number of rows in the matrix [9].
- Confirming that each row was generated correctly as per the generic generator matrix shown above.

4.2 The Simulation Script

We were then able to use the theory discussed in the Overview section and implement a program that ran for various rate $^1/_n$ convolutional codes. The function of the program was to simulate the writing and reading of data to/from a SSD using the encoding process discussed in Section 3.4.

Our simulations required pre-defining the codeword length, number of levels in a flash cell, convolutional code rate, and constraint length in order to be run. The program simulated writes and reads to one page of data, and wrote to the page in chunks. The size of a chunk was the length of a dataword; this can be calculated using Equation 4.1 for a given codeword length and convolutional code rate. Once a chunk was encoded and written to the SSD, we would then make sure that all writes were successful by looking at the charge level of each flash memory cell. On an unsuccessful write to a cell (writing to a saturated cell), the program would log the error for that chunk and stop any subsequent writes after attempting to write the remaining chunks. In order to verify the correct operation of the simulation, we would decode all successfully written chunks and ensure they were the same as the original chunk.

For our simulations, we used a codeword length of 1024 bits for each chunk of data. Simulations were run to determine the number of writes before failure for rate ½ and ¼ codes while sweeping constraint length and MLC levels.

In order to be tested on hardware, the dataword would still be encoded on a computer before being written to a physical SSD. Therefore, the same program could be used in running tests for hardware.

4.3 Applying Different Error Metrics in the Viterbi Algorithm

We implemented different metrics for Viterbi Decoding in order to try and improve the write efficiency for a rate ¼ code. Using different metrics allows us to change the best path chosen in encoding data using Viterbi decoding (refer to Section 3.3 for more detail). The goal behind creating each metric was to improve wear-leveling i.e. evenly distributing writes across each set of bits outputted in Viterbi decoding. Since these improvements were being made for a rate ¼ code, the metric was evaluated for each edge outputted by the Viterbi algorithm during each time-step. For each metric implemented, we record a wear leveling benchmark (Equation 4.2) and the number of successful writes before failure.

$$Wear\ Leveling\ Benchmark = \frac{\frac{\#\ of\ Failed\ Cells + \#\ of\ Saturated\ Cells}{\#\ of\ Failed\ Chunks}}{Encoded\ Chunk\ Size} \tag{4.2}$$

The closer the benchmark is to 1, the more wear-leveled the encoded page is on failure.

These metrics were tested using constraint length equal to 6, with 2-level cells and 4-level cells. The reason behind using a constraint length of 6 is because using higher constraint lengths results in longer encoding time for a dataword that could bottleneck the write bandwidth of the SSD. The encoded chunk size being used was 1024 bits. Below, we describe the details of each metric.

- **Bit-Flip Reduction**: This method involves flipping the least number of bits possible when writing new data over data that is currently stored in the cell. Each bit, to be written, is assigned a certain weight (known as a bit flip weight); if a bit to be written is different from the bit already stored, then this weight is added to the resultant metric. If the bit to be written is the same as what is already stored, then a minimum penalty (generally 0) is added to the resultant metric for that edge. While bit-flip reduction is simple, it proves to be effective by avoiding the saturation of a cell. Cells closer to saturation have higher bit-flip weights thus decreasing the chance of that edge being included in the final path for encoding.

- **Adding Bit-Flip Weights**: As mentioned above, each bit, to be written, is assigned a certain weight (or penalty) that is added to the metric if the bit is flipped. The weight of a cell increases, as it gets closer to saturation, therefore summing the bit-flip weights is a decent indicator of how saturated a set of cells to be written is; a set of less saturated cells will have a lower bit-flip weight. Here, we add the bit-flip weight of each cell to be written regardless of what the bit to be stored is.

- **Adding Bit-Flip Weights + Bit-Flip Reduction:** This is a combination of the above two metrics i.e. we add all the bit-flip weights, and then further penalize a cell which requires a bit to be written that is different from what is stored.

- **Applying a Negative Bit-Flip Minimum Penalty:** In the bit-flip reduction metric, we talked about a minimum penalty applied to bits that didn't require flipping during a write. In general, we assigned a minimum flip penalty of 0, but we tried assigning a negative minimum penalty to further reward an edge if no bit was flipped. The penalty rewarded was -1.

- **Changes in Flash Cell Levels per Edge:** The more wear-leveled an edge is the less changes it has in its values. Using this property, we add the bits we wish to write to that edge and then calculate the cumulative change in consecutive values to determine how wear-leveled an edge would be had we written those bits to it.

5 Results and Analysis

5.1 Simulation Results

First, we ran simulations for a rate ½ convolutional code and a rate ¼ code under the same parameters. The Viterbi metric being used was the same as what was being used by Jacobvitz et al. i.e. the Hamming distance between two edges. The simulations run for each convolutional code measured the number of writes to a page before failure, while varying constraint length and MLC levels. This can be seen in Figures 5.1 and 5.2.

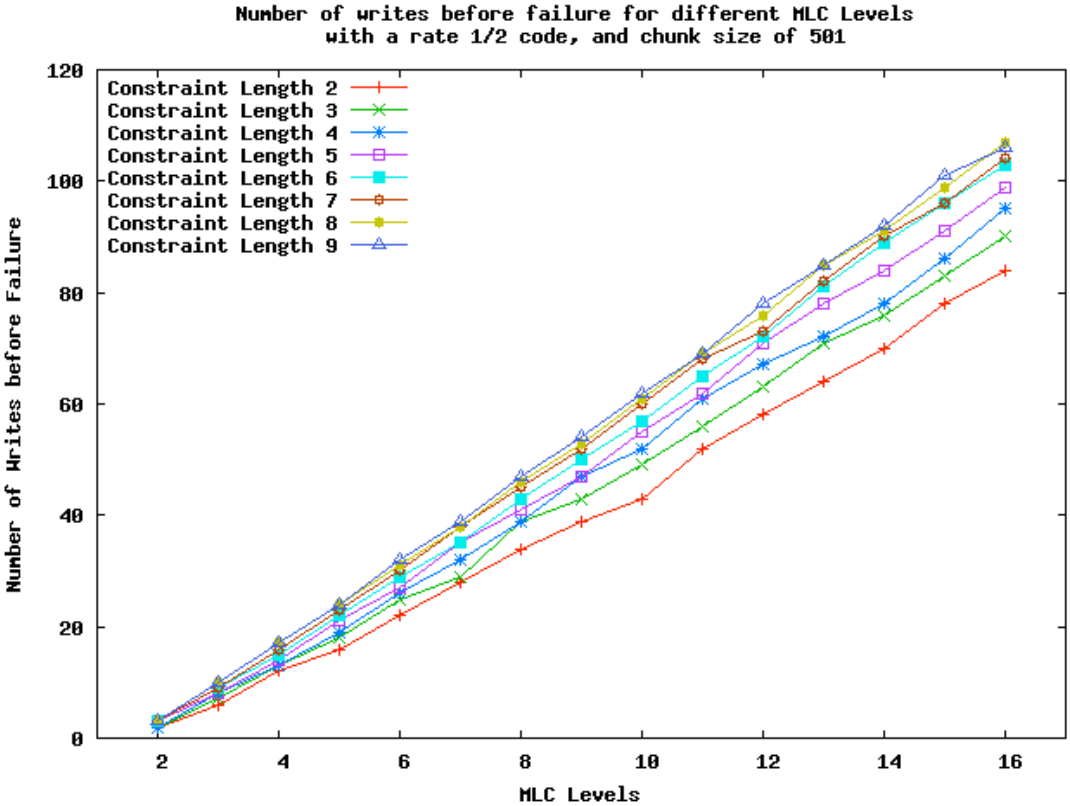

Figure 5.1 – Number of Writes for a Rate ½ Code

As expected, the number of writes increases with number of levels in a flash cell as well as with the constraint length. It is also interesting to note that the number of writes per constraint length spreads out more with higher level flash cells. The maximum number of writes is achieved is 106 writes before failure using 16-level cells and a constraint length of 9. While we see a drastic improvement in the number of writes for a rate ½ code, it comes at the cost of a 100% area overhead, which is greatly undesirable. Therefore, we look at simulations for a rate ¼ code.

Leaving all conditions the same as a rate ½ code, we then ran simulations using a rate ¼ code. The results can be seen in Figure 5.2.

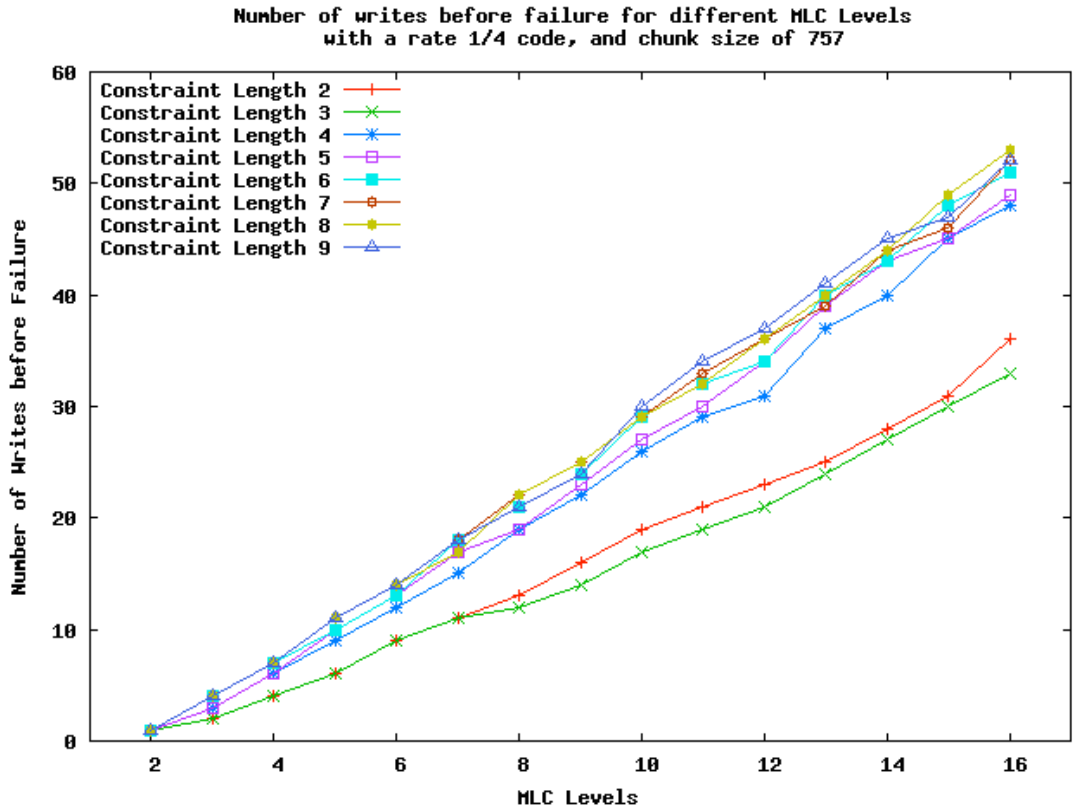

Figure 5.2 – Number of Writes for a Rate ¼ Code

Comparing Figure 5.1 and 5.2, we see that the number of writes using a rate ¼ code for a given constraint length and flash level cell is about half the number of writes completed using a rate ½ code under the same conditions. The reason for the reduction in number of writes is because the trellis constructed by the Viterbi algorithm explores fewer cosets for a rate ¼ code than a rate ½ code. This means we have fewer alternative representations of the dataword resulting in a less-optimal encoded output.

We also notice that the number of writes with higher-level cells increases drastically for constraint lengths 3 and above. We do not see such an increase with a rate ½ code. The reason for this difference between both codes is most likely to do with the metric being used for each edge. Since the edge for a rate ¼ code has 4 bits, it allows for a more flexible metric than a rate ½ code. This flexibility could be the reason for the sudden jump in number of writes.

For both the rate ½ code and rate ¼ code, the number of writes before failure increases linearly with MLC levels. This is not what we expect because adding a flash level should result in a more exponential growth in number of writes. We believe the reason behind this is poor wear leveling and hence we look further into that.

Furthermore, we see that a rate ¼ code has about a 100% improvement in number of writes for 4-level cells, but has no improvement in number of writes for 2-level cells. This lack in improvement could render a rate ¼ code useless for 2-level cells and thus it is important for us to further examine how we can increase the number of writes for 2-level cells.

5.2 Applying Different Error Metrics in the Viterbi Algorithm

The goal behind modifying the error metric was to improve the wear leveling of each edge. By changing the metric, we change the best path that the Viterbi algorithm chooses for writing an encoded dataword to the SSD. The different metrics applied are described in Section 4.3. For each metric, we recorded the number of writes before failure and a wear leveling benchmark (Equation 4.2). Simulations were only run for a rate ¼ code because we were focused at improving its write efficiency. The results for applying each metric can be seen in Table 5.3.

Table 5.3 – Performance of a Rate ¼ Code Using Different Error Metrics

		2-Level Flash Cells		4-Level Flash Cells	
#	Metric	Wear Leveling Benchmark	Number of Writes before Failure	Wear Leveling Benchmark	Number of Writes before Failure
1.	Bit-Flip Reduction	0.54	1	0.29	7
2.	Adding Bit-Flip Weights	0.73	1	0.30	3
3.	Adding Bit-Flip Weights + Bit-Flip Reduction	0.54	1	0.29	7
4.	Applying a Negative Bit-Flip Minimum Penalty	0.54	1	0.31	7
5.	Changes in Flash Cell Levels Per Edge	0.89	1	0.63	3

Looking at the above table, we see that metric 5 does the best in wear leveling as it achieves the highest wear leveling benchmark for both 2-level and 4-level flash cells. For 4-level flash cells however, we have a significant reduction in the number of writes when using metric 5. As for 2-level flash cells, the number of writes is the same across all metrics and hence we believe that metric 5 might actually have the highest potential in increasing the number of writes to 2 writes before failure.

Examining the results for 4-level cells closely, we see that metrics 2 and 5 have significantly less writes than the others. This is interesting because we expected them to perform the best wear leveling and hence get the highest number of writes. While metric 5 holds up to expectations with wear leveling, we were surprised by the reduction in number of writes. Ironically, metric 1 has the highest number of writes though it has the least focus on wear leveling i.e. it only tries to avoid saturating a cell. Metrics 3 and 4 both incorporate bit-flip reduction and also have the same number of writes as metric 1. On running the simulations for 6-level cells, we saw that metric 1 still obtains the highest number of writes. This adds further credibility to a hypothesis that bit-flip reduction is essential in obtaining the most writes.

6 Conclusion

Through our simulations for a rate ¼ convolutional code, we see that it provides no improvement in write efficiency for 2-level flash cells and about a 100% improvement for 4-level flash cells. These improvements come at a cost of a 35% area overhead for the SSD. Since 2-level and 4-level cells are the most commonly used flash cells in the industry, we need to show higher improvements in write efficiency if we want to justify the area overhead incurred by the convolutional code.

In our efforts to improve the write-efficiency, we applied different error metrics to be used in the Viterbi algorithm. While none of the applied metrics resulted in an increase in writes, we were able to determine that bit-flip reduction played a crucial role in maintaining a higher write efficiency. In addition to this, we saw that examining changing values across an edge resulted in better wear leveling, but was unable to get a high enough write efficiency. Future work might include looking at the optimal combination between wear leveling and bit-flip reduction such that we can increase the write efficiency and justify the use of a rate ¼ convolutional code in industry standard SSDs.

7 Acknowledgements

I would like to thank Professor Sorin for providing me with the opportunity to conduct research in this area and for his mentorship over the past two years. A special thanks to Adam Jacobvitz for taking the time out to teach me all the theory and help set up the program for running all the simulations. Adam was incredibly supportive throughout the entire research period, and offered great insight when I would run into roadblocks. I would also like to thank Jiayu Gong for his help in running the simulations and identifying possible flaws in the system.

8 References

[1] A. N. Jacobvitz, A. R. Calderbank, and D. J. Sorin, "Writing Cosets of a Convolutional Code to Increase the Lifetime of Flash Memory," in *Proceedings of the 50th Annual Allerton Conference on Communication, Control, and Computing*, 2012.

[2] D. G. Andersen and S. Swanson, "Rethinking Flash in the Data Center," *IEEE Micro*, vol. 30, no. 4, pp. 52– 54, Jul. 2010.

[3] *International Technology Roadmap for Semiconductors, 2011 Edition, Process Integration, Devices, and Structures.* 2011.

[4] X. Y. Hu, E. Eleftheriou, R. Haas, I. Iliadis, and R. Pletka, "Write Amplification Analysis in Flash-based Solid State Drives," in *Proceedings of SYSTOR*: *The Israeli Experimental Systems Conference*, pp. 10:1–10:9, 2009.

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

[6] A. Viterbi, "Error Bounds for Convolutional Codes and an Asymptotically Optimum Decoding Algorithm," *IEEE Transactions on Information Theory*, vol. 13, no. 2, pp. 260 –269, Apr. 1967.

[7] Hamming, "Error Detecting and Error Correcting Codes," *Bell System Technical Journal*, vol. 29, no. 2, pp. 147–160, Apr. 1950.

[8] Moore, "Cramming More Components onto Integrated Circuits," *Electronics Magazine*, pp. 114-117, Apr. 1965.

[9] Lidl, Rudolf, Niederreiter, Harald, *Finite fields*, Encyclopedia of Mathematics and Its Applications 20 (2nd ed.), 1997.

[10] 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.

[11] M. Moshayedi and P. Wilkison, "Enterprise SSDs," *ACM Queue*, vol. 6, no. 4, pp. 32–39, Jul. 2008.