This paper appears in the 19th IEEE International Symposium
on High Performance Computer Architecture

Coset Coding to Extend the Lifetime of Memory

Adam N. Jacobvitz
Dept. of ECE
Duke University

Robert Calderbank
Depts. of ECE, Math, and CS
Duke University

Daniel J. Sorin
Dept. of ECE
Duke University

Abstract

Some recent memory technologies, including phase change memory (PCM), have lifetime reliabilities that are affected by write operations. We propose the use of coset coding to extend the lifetimes of these memories. The key idea of coset coding is that it performs a one-to-many mapping from each dataword to a coset of vectors, and having multiple possible vectors provides the flexibility to choose the vector to write that optimizes lifetime. Our technique, FlipMin, uses coset coding and, for each write, selects the vector that minimizes the number of bits that must flip. We also show how FlipMin can be synergistically combined with the ability to tolerate bit erasures. Thus, our techniques help to prevent bits from wearing out and can then tolerate those bits that do wear out.

1. Introduction

Some non-volatile memory technologies, including phase change memory (PCM) and Flash, have lifetime reliabilities that are affected by write operations. Ideally (and unrealistically), we would like to completely control the bits being written to memory such that we maximize lifetime. That is, *regardless* of what the core wants to write, we would be able to choose what we write to memory in order to maximize lifetime.

Clearly, there must be some connection between the *dataword* that the core wants to write and the *vector* of bits that is written to memory. In current memory systems, this connection is highly constrained. When a core wishes to write a dataword to memory, it writes a vector that is the dataword plus perhaps some bits for error detection or correction. There is a one-to-one mapping between the dataword and the written vector.

In this paper, we propose using a technique from coding theory, called *coset coding*, to provide flexibility in mapping from the dataword to the vector. Using coset coding (for which we provide a brief tutorial in Section 3), we enable a one-to-many mapping instead of the one-to-one mappings currently used. Coset coding is a general technique that is parameterizable such that we can trade off flexibility (i.e., how many possible vectors map to the same dataword) versus cost (i.e., how many extra bits are required for the written vector compared to the dataword).

The ability to choose a vector from among a set of possibilities allows us to choose the newly written vector so as to optimize objectives. In this paper, we choose the vector so as to minimize the number of bits that must change from the vector previously written to that location. This optimization, which we call *FlipMin* (and discuss in Section 4), extends memory lifetime by reducing the number of bits flipped during the lifetime of the memory. Conversely, FlipMin can also be used to achieve the same lifetime while reducing the cost of producing memory by tolerating greater manufacturing variances.

This paper makes three primary contributions:

1) We show how to use coset coding to prolong the time before bits wear out by minimizing the number of bits that flip per write.
2) We present a method to synergistically combine FlipMin with the ability to tolerate bit erasures when wearout does occur.
3) For a subset of possible coset codes we can use with FlipMin, we evaluate how well the codes extend the lifetime of PCM for both random and benchmark inputs. For the same coset codes, we provide example encoder/decoder hardware and evaluate it in terms of energy, area, and performance overheads.

2. Related Work

Existing schemes for extending the lifetime of write-limited memories can be grouped into four broad categories. We list prior schemes in Table 1, and we shade schemes that we do not quantitatively compare against in Section 7.

2.1. Postponing Wearout: Bit Flip Reduction

All else being equal, if fewer bits flip per write to a memory location that location will last for a greater number of writes. Bit flip reduction, for the purpose of postponing wearout, is the technique we use in this paper. To the best of our knowledge, the only other work in this area is Flip-N-Write [5]. At each write, Flip-N-Write chooses to write the dataword or its inverse, depending on which requires fewer bit flips. Flip-N-Write adds a single bit per location to indicate whether the data is inverted or not. We will show later

Table 1: PCM lifetime extension schemes. We quantitatively compare to un-shaded rows.

Approach	Scheme	Instantiation	Granularity	Overhead	Why No Quant. Comparison
bit flip reduction	Flip-N-Write (FnW) [5]	FnW per-byte	8 bits	1 bit=12.5%	
		FnW per-word	64 bits	1 bit=1.56%	subsumed by FnW per-byte
	Coset Coding	*discussed in paper*	64 bits	*tunable*	
error/erasure correction	ECC	Hamming (72,64)	64 bits	8 bits=12.5%	
	ECP [23]	ECP_6	block ~ 512 bits	61 bits=11.9%	
		ECP_{12}	block ~ 512 bits	121 bits=23.6%	
		ECP-ideal	block ~ 512 bits	0	
	Pay-As-You-Go [19]		entire memory	*tunable*	subsumed by ECP-ideal
	SAFER [25]	SAFER8	block ~ 512 bits	22 bits=4%	subsumed by ECP-ideal
		SAFER32	block ~ 512 bits	55 bits=10.7%	subsumed by ECP-ideal
	RDIS [16]	RDIS3	block ~ 512 bits	*see* †	subsumed by ECP-ideal
	FREE-P [27]		block ~ 512 bits	64 bits=12.5%	requires OS support
	DRM [9]		page ~4KB	*see* ǂ	requires OS support
adding cells	DoubleMem		64 bits	64 bits*=100%	

† *Overhead is listed as 18%, but RDIS does not account for overheads to track erasures.*

ǂ *12.5% to track erasures plus 100% for paired pages plus a single 1KB "ready table"*

* *Actual overhead is greater than 64 bits due to extra state bits to track which copy of the location is being used.*

that Flip-N-Write is a degenerate instance of FlipMin. Because Flip-N-Write is the only prior work with the same goal as FlipMin, our experiments focus on comparing them. We assume for all schemes that the hardware only writes to those bits whose values change [26], i.e., there is no wearout incurred by writing a bit if the bit's value does not change.

One other way to minimize bit flips is to coalesce multiple writes [15] before applying them to the PCM. Write coalescing is a useful technique that is orthogonal to all prior work and to our work.

2.2. Tolerating Wearout: Error Correction

The other dominant technique for extending memory lifetime is to tolerate bit errors after wearout occurs. Tolerating wearout is effective when a minority of cells at some location granularity (e.g. byte, line, etc.) fail far earlier than average, making the entire location unusable. Example schemes include error correcting codes (ECC) and some techniques specific to write-limited memories [9][16][19][23][25][27]. One prominent scheme that we compare against quantitatively is Error Correcting Pointers (ECP) [23]. The ECP scheme tolerates errors in known bit positions (i.e., *erasures*) in memory locations by maintaining pointers to these bit positions and adding bits to be used as replacements. For example, ECP_6 operates at a 512-bit location granularity, and it keeps six 9-bit pointers ($\log_2(512) = 9$) and 6 replacement bits for tolerating up to 6 erasures in the 512-bit location.

There has been a large amount of work that extends and optimizes ECP, including Pay-As-You-Go [19], SAFER [25], and RDIS [16]. Rather than compare against all of them, we compare to an idealized and un-implementable ECP_{12} that has zero cost, which we refer to as ECP-ideal. For our particular experimental purposes, ECP-ideal subsumes the work that reduces ECP's costs. FREE-P [27] and DRM [9] are two other techniques for tolerating erasures. Both of them require OS support, which we do not assume in our work. Furthermore, direct comparisons to FREE-P are difficult to make fair because of FREE-P's added support for tolerating soft errors.

2.3. Bit Flip Reduction + Error Tolerance

We can combine bit flip reduction schemes with error/erasure tolerance schemes to achieve the best of both worlds. For example, FlipMin can be combined with ECP. Such hybrid schemes both postpone wearout and tolerate it when it eventually occurs.

2.4. Adding Memory Cells

One can extend the lifetime of write-limited memories by simply adding more memory cells and using these cells for purposes of extending lifetime rather than increasing logical capacity. For example, if a memory location is logically 64-bits, we can use 128 physical bits in a scheme we call DoubleMem. With DoubleMem, initially the first 64 bits of the physical location are used to store data. When the first 64-bit physical location fails, the second 64-bit physical location is used to store data. There has been research into more sophisticated methods for adding memory cells to improve lifetime, including waterfall codes and hypercells [14], but they all share the same idea.

2.5. Wear-leveling

Intra-location wear-level schemes try to level out the wear in a given location more uniformly (e.g., by remapping logical bit positions) [12][29]. These schemes require state to track the current bit position mappings for each location, and they require sophisticated heuristics to decide when and how to remap bit positions.

Inter-location wear leveling schemes seek to avoid writing to some locations more frequently than others. These schemes [12][29][21][24][20] avoid these

situations by dynamically mapping from logical locations to physical locations, in ways that are similar to, but simpler than, virtual memory's translations from virtual pages to physical pages. This work is complementary to our work.

3. Coset Coding Primer

The key enabling technology is the use of coset coding [6][7]. In this section, we explain coset coding, starting with the basic idea (Section 3.1) and a simplistic example of a coset coding scheme (Section 3.2). We then present a more realistic coset coding scheme (Section 3.3). We defer a discussion of implementation issues until Section 6.

3.1. Basic Idea

Consider a k-bit dataword. This dataword can be any element of a set of 2^k possible strings of zeros and ones. We map datawords from the set of 2^k possibilities to a larger set of 2^n possible n-bit vectors ($n>k$). One well-known example of a one-to-one mapping is single-bit parity. For example, even parity maps each dataword in a 2^k set to a vector in a set of size 2^{k+1}, such that each of the mapped 2^{k+1} vectors has an even number of ones.

Coset coding extends this basic concept to a one-to-many mapping, illustrated in Figure 1. The 2^n set of vectors is split into non-overlapping, equal-sized *cosets*. A coset is a set that is generated in a very specific way such that every coset is disjoint from every other coset and the union of the cosets covers a group (e.g., the set of all possible n-bit vectors under the XOR operator). Formally defining cosets is more likely to confuse than enlighten; we instead highlight cosets' relevant properties as needed.

In general, if $c = n - k$ (i.e., c is the number of additional bits needed for a given vector in a coset compared to a dataword), then the 2^n set of vectors is divided into 2^{n-c} cosets, each with 2^c elements. Now we can choose n and c such that $k = n - c$. That is, the number of cosets (2^{n-c}) equals the number of datawords (2^k), and each dataword maps to a single coset.

The key idea is that, for a given dataword, we can dynamically choose which element of that dataword's coset to write. Because each coset has 2^c elements, we have 2^c options for which vector to write. When we read from a location, we obtain an element of a coset, determine which coset it belongs to, and perform the one-to-one mapping from that coset to the dataword.

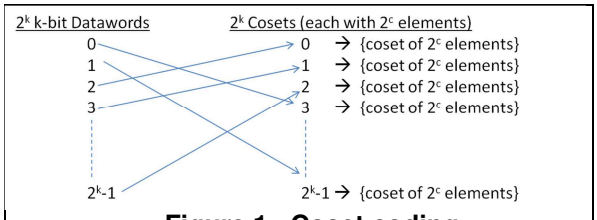


Figure 1. Coset coding

The benefits of coset coding derive from the flexibility to choose which element to write so as to optimize an objective. For example, if we know the previously written vector to be overwritten, we can choose the new element to minimize the number of bits that must flip.

The primary cost of coset coding is its storage overhead of c bits per vector. Compared to simply storing the original dataword of k bits, we now store an n-bit vector ($n=k+c$). There is a fundamental tradeoff between the benefit and cost of coset coding. To increase the benefit requires mapping each dataword to a coset with more elements in it. Increasing the number of elements per coset requires increasing n, and increasing n increases the cost.

Coset coding also incurs a small computational cost for encoding and decoding, which we discuss later.

3.2. Simplified Coset Coding Example

To make the idea of coset coding more concrete, we provide a simplistic example. Assume datawords are 2-bit values (denoted D1 through D4) and that we map each dataword to a coset that consists of four 4-bit vectors. We divide the set of all possible 4-bit vectors into 4 cosets (denoted S1-S4), as follows:

D1=00 maps to S1 = {0000, 0101, 1010, 1111}
D2=01 maps to S2 = {0001, 0100, 1011, 1110}
D3=10 maps to S3 = {0010, 0111, 1000, 1101}
D4=11 maps to S4 = {0011, 0110, 1001, 1100}

Thus, if the core wishes to store the value 01 (dataword D2), it can choose to write any of the elements from coset S2. If the core reads from memory and obtains the vector 1001, which is an element in S4, the core interprets it as 11, because dataword D4 (11) maps to S4. Notice that the cosets are entirely non-overlapping; no vector appears in more than one coset. This property is crucial for reading from memory, because reading requires mapping uniquely from a written vector to a dataword.

Overwriting an existing vector with a new vector involves flipping a number of bits that is a function of the cosets to which the existing and new vectors belong. For example, if the existing written vector is 0101 (in coset S1) and the new vector to write is in coset S2, then we can minimize the number of the bits flipped by choosing vector 0100 from S2.

3.3. Practical Coset Coding with Block Codes

The simple coset coding scheme in the previous section has two drawbacks. First, its distribution of elements within each coset does not actually enable us to reduce the number of bits flipped. Second, the division of the set of vectors into cosets (called *coset generation*) was done manually, which is infeasible for larger sets. We now describe automatic coset generation and how to encode/decode.

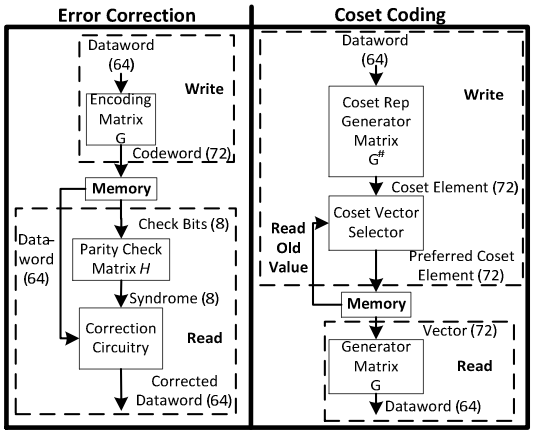


Figure 2. Hamming(72,64) for Error Correcting (left) and Coset Coding (right)

3.3.1 Coset Generation

To automate coset generation, we use *linear algebraic block codes (LABC)*. There are many possible block codes that would serve this purpose, and they offer different cost/benefit tradeoffs. LABCs are often used for purposes of error detection and correction. One such code is the commonly-used Hamming (72,64) code that maps 64-bit datawords into 72-bit codewords. In a typical use of a Hamming code for error correction (Figure 2, left), a codeword is created by multiplying a dataword, $\vec{d}$ with the code's *generator matrix*, *G*. A codeword, $\vec{c}$, is decoded by multiplying the codeword with the code's *parity-check matrix*, *H*. If the result of $H\vec{c}$, which is called the *syndrome,* equals the zero vector, then $\vec{c}$ is error-free.[1] If there was a single bit error, the syndrome identifies which bit in the dataword is erroneous; flipping that bit corrects the error.

For coset generation, we use LABCs differently than how they are used in error correcting codes. Consider the Hamming (72,64) code. There is a one-to-one mapping between each 64-bit dataword and its corresponding (error-free) 72-bit codeword. We then have an 8-bit syndrome that, when it is all-zero, indicates the codeword was transmitted correctly. There is a many-to-one (e.g., 2^{64}-to-one for Hamming(72,64)) mapping of vectors that result in the same syndrome. The sets of vectors that all map to the same syndrome perfectly partition the vector space into cosets. We will refer to the coset of codewords (all codewords map to the all-zero syndrome) as the *zero coset*, because it has some properties we exploit later.

We could use Hamming(72,64) in this way to generate 2^8 cosets, each of which has 2^{64} elements; however, we would rather have more cosets and fewer elements per coset. One way to achieve this goal is to use the code's *dual code* (which is also a block code with its own zero coset), in which we flip the uses of the matrices *G* and *H*. Hamming(72,64)'s dual code maps 8-bit datawords to 72-bit codewords, and syndromes are 64-bits. Its dual code partitions the 72-bit space into 2^{64} cosets, each with 2^8 elements.

Coset generation with block codes has several attractive properties, two of which we will exploit in this work. First, the zero coset is a *group* under bitwise addition modulo 2 (i.e., the bit-wise XOR operation). That is, if we take any two elements from the zero coset and XOR them together then we obtain an element from that same coset. Furthermore, XORing any element from the zero coset with any element in coset *X* produces an element in coset *X* (which is why one of the 2^c elements in the zero coset is all-zero). We will later use the zero coset to generate the elements of a coset starting from just one element in the coset, by XORing the element from the given coset with every element of the zero coset. Second, we can *translate* any element from a coset *X* to some element in another coset *Y* by XORing the element from *X* with any element from another coset called the *translate coset T*. (Each *X,Y* pair has a corresponding *T* that is just another coset in the same space. *T* is the zero coset if *Y*=*X*.) That is, if we want to overwrite an element in *X* with an element in *Y*, we can XOR the element in *X* with any element in *T*. We can then choose the element from *T* to minimize the number of bits that must flip. If we know the identities of *X* and *Y* and we know the zero coset, we can easily determine *T* (explained in more detail in Section 6).

Let us return to the simplistic coset coding scheme from Section 3.2. Consider the case in which the currently written vector is 1010 (an element in coset *S1*), and we wish to write dataword *D2* (which corresponds to *S2*). In the table below, we show the four options for translating from 1010 to elements in *S2*. The four elements in the translate coset are in the bottom row. We can choose any of these translate elements, based on the objective we seek to optimize. For example, we might choose Option #3 (translate element 0001), because it requires the fewest bits to flip (i.e., translate element 0001 has the fewest ones).

	Option #1	Option #2	Option #3	Option #4
Vector Written Prev	1010	1010	1010	1010
Elements in S2	0001	0100	1011	1110
Translate Element	1011	1110	0001	0100

3.3.2 Writing (Encoding) and Reading (Decoding)

We present the high-level view of how writing (encoding) and reading (decoding) work, and we defer the implementation details to Section 6. We illustrate the processes in Figure 2 (right), and we note that they are similar to those for typical ECC. The three differences are: the use of different matrices, the reading

[1] Hamming(72,64) has a non-zero syndrome for 1-bit or 2 bit-errors.

Table 2. FlipMin with different block codes

Block Code	Storage Overhead per 64-bit dataword	Bit Flip Reduction (compared to unencoded)	Comments
dual of Parity(72,64)	8/64 = 12.5%	15.8%	Performed on 8-bit chunks
dual of RM(1,7)T	8/64 = 12.5%	24.5%	
RM(1,3)	64/64 = 100%	31.2%	Performed on 4-bit chunks

of the prior vector before writing the new vector, and the logic to choose a preferred element of a coset.

4. FlipMin: Extending Lifetime by Reducing Bit Flips

Our application of coset coding, which we call FlipMin, is to extend the lifetime of memory by minimizing the number of bit flips per write. The goal is to choose the new vector to be written that is most similar to the vector already written to that location.

4.1. How It Works

The key idea of FlipMin is to choose the translate element within the translate coset *T* that has the least *weight* (i.e., the fewest number of bits that are one); this element is called the *coset leader* of *T*. Some cosets have multiple coset leaders, in which case we can choose any of them. To overwrite the currently written element in *X* with an element from *Y*, FlipMin XORs the element from *X* with the coset leader of *T*. Because the coset leader has the fewest ones of any element in the translate coset, FlipMin thus flips the fewest possible bits. The extent to which FlipMin reduces bit flips depends on the particular code used.

4.2. Practical Block Codes for Coset Coding

There are a vast number of possible block codes that can be used for FlipMin. We have already mentioned Hamming(72,64) and its dual. We also looked at the simplest code for FlipMin, parity. Parity FlipMin is equivalent to an already published scheme, Flip-N-Write [5]. There are numerous other codes that work with FlipMin—including other Hamming codes, BCH codes, and Reed-Muller (RM) codes [22]—and they represent different trade-offs between storage overhead and flexibility.

We assume datawords are 64-bits long; for those codes that map datawords smaller than 64-bits, we divide the dataword into chunks and encode each one independently. For example, with Reed-Muller(1,3) (also known as Hamming(8,4)), we encode a 64-bit dataword by encoding it in 16 4-bit chunks. In Table 2, we list the codes we consider in this paper. We evaluated parity coset coding at the 8-bit granularity giving it 12.5% overhead. Reed-Muller codes, including their dual codes, have codeword lengths that are powers of two, so we truncated[2] the dual of the RM(1,7) code, denoted RM(1,7)T, to reduce the overhead for 64-bit datawords from 100% to 12.5%.

[2] Truncation is a common technique for modifying a code's length.

4.3. Analytical Evaluation of FlipMin

For purposes of this exposition, consider a single memory location and 4-bit datawords. Furthermore, for now, assume there is a stream of random datawords to be written to this memory location.

In a system without coding of any kind (i.e., the vector written is the dataword itself), an average of half of the bits (2) within a memory location will flip per write. Intuitively, for random datawords, each bit is equally likely to flip due to a write. However, it is instructive to more thoroughly reason about this relationship, because it will help to understand the analysis of FlipMin. In Table 3, the leftmost pair of columns shows the histogram of dataword *weights*, where the weight of a word is the number of ones in it. Using a weighted average of the dataword weights, we obtain: $(1\times0 + 4\times1 + 6\times2 + 4\times3 + 1\times4)/16 = 2$. This weighted average is another way to arrive at the average number of bits that will flip per write.

In Table 3, we show FlipMin's impact, given a random stream of inputs, and we show results for RM(1,3), which is the simplest to illustrate. The number of bits that flip, on average, is a weighted average of the weights of the translate cosets. The results show that, using coset coding, we dramatically reduce the average number of bit flips from 2 down to 1.375, which is a 31% reduction.

For block codes other than RM(1,3), it is impractical to enumerate all of the translate coset weights. Instead, in Table 2, we simply present the computed reduction in bit flips as a function of the block code, once again for a random stream of writes.

From the analytical results, we conclude that FlipMin can extend memory lifetime by reducing the number of bit flips. Furthermore, there is great flexibility in this approach, because we can choose among many different block codes. These analytical results are a function of the datawords to be written, which in this case is

Table 3. Average number of bits written

Unencoded 4-bit dataword		FlipMin with RM(1,3)	
word weight	# of words	translate coset weight	# of words
0	1	0	1
1	4	1	8
2	6	2	7
3	4	3	0
4	1	4	*0*
Avg bit writes	*2.0*	*Avg bit writes*	*1.375*

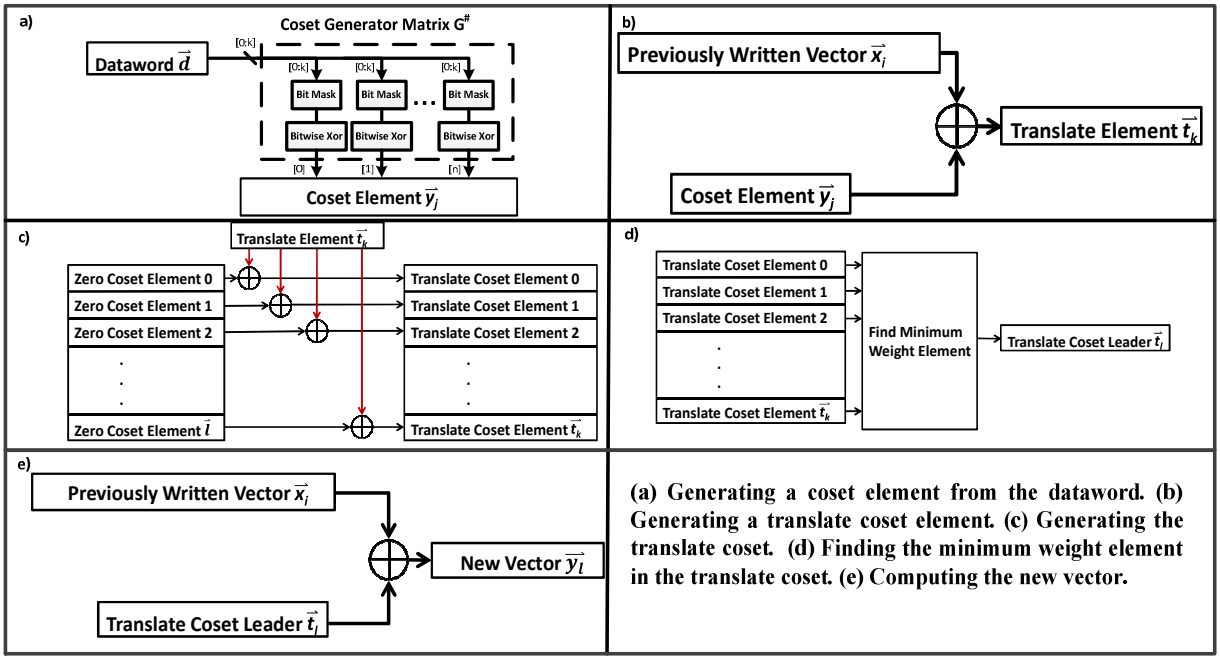


Figure 3: Hardware for encoding.

assumed to be random. In Section 7, we will experimentally evaluate FlipMin with both random traces and traces from benchmarks.

5. Coset Erasure Matching (CEM)

In addition to reducing bit flips, coset coding can also tolerate *erasures* (defects in known bit positions). There has been theoretical research on supporting erasures using coset coding [13] and more specific theoretical studies on coset coding showing the feasibility to tolerate erasures in Flash [10] and PCM [11].

Our particular implementation we call Coset Erasure Matching (CEM). When it is time to pick which element of a coset to write, our selection algorithm chooses an element that accommodates the erasure(s). For example, assume that a given memory location has bit *B* stuck-at-1. When writing to that location, FlipMin can choose to write an element from the appropriate coset such that the chosen coset element has a 1 at bit position *B*. The erasure is thus tolerated, but at the cost of limiting the scheme's flexibility in choosing an element from the appropriate coset.[3] To tolerate erasures, we must both identify the erasures and then maintain state that identifies the defective bits.

6. Implementation

In this section, we discuss hardware implementations for writing (encoding) and reading (decoding).

[3] CEM tolerates erasures by choosing randomly from among the elements that are compatible with the erasure(s).

6.1. Encoding

Assume that the memory location currently holds a vector x_i that is an element in coset *X*. There are five steps to encoding. First (Figure 3a), we map $\vec{d}$ to a to an element y_j in coset Y. One was to do this mapping is to use a coset generator matrix, $G^{\#}$, with which we multiply (modulo 2) the dataword. For a block code, we can form $G^{\#}$ by taking the left inverse of the block code's generator matrix, G [8]. Other solutions include a lookup table or using a trellis [4]. Second (Figure 3b), we perform $\vec{x_i}$ XOR $\vec{y_j}$ to obtain an element $\vec{t_k}$ from the translate coset T. Third (Figure 3c), we generate the rest of T by XORing $\vec{t_k}$ with every element in the zero coset. Fourth (Figure 3d), we choose the specific element within T that optimizes our objective function. Fifth (Figure 3e), we perform $\vec{t_l}$ XOR $\vec{x_i}$ to get the new element $\vec{y_l}$ that we then write to memory.

6.2. Decoding

Decoding is the simpler of the two operations, which is fortuitous because decoding (reading memory) is generally more critical to performance than encoding (writing memory). To decode a written vector, we multiply it by the block code's generator matrix, *G*.

One possible hardware implementation for this multiplication is shown in Figure 4. The written vector passes through *k* bit-masks which are hardwired to select wires based on *G*. Each *n*-bit output of each bit-mask is then bitwise XORed to produce one of the *k* bits of the dataword.

6.3. Overheads

We used Synopsys Design Compiler and IC Compiler to synthesize, layout, and floorplan encoders and decoders for different coset codes on top of the Nangate 45nm semi-custom cell library [18]. For energy evaluation we generated 1000 random inputs. We then simulated each input using VCS to back annotate a SAIF file for the activity factor. We then used Design Compiler Topographical for energy evaluation using the generated SAIF file.

Our coset coding encoders take 64-bit inputs and our coset coding decoders produce 64-bit outputs. To compare to the overhead of the PCM chip [17], which has 16-bit I/O width, we multiply the area and power overheads from the PCM datasheet by 4.

6.3.1 Delay

The maximum delay among the encoders, as shown in Table 4, is 12.86ns. The PCM write time is 60-120 µs. Thus the additional write time imposed by our encoder, even in the worst case, is negligible. From the same PCM datasheet we have a read time of 115ns + 25ns per 16-bit entry. Our worst case decode time is 0.59ns, which again is low enough to be in the noise of the read times.

6.3.2 Energy Consumption

The average energy listed in Table 4 and Table 5 is the average of the energy values we found for 1000 simulations. The worst case energy is the maximum of the values we simulated. We found the worst case energy consumption to be 63.4 pJ. The typical idle current of 4 PCM devices is around 320µA and the minimum rail voltage is 1.7V. Thus the minimum idle power is about 544µW. If we encode a block every 60µS, the fastest write time possible for the PCM chip, our FM-RM(1,7)T encoder would take 1.06 µW, which is negligible compared to the idle power of the PCM chip. The decoders for coset coding take much less energy than the encoders (the worst case decoder energy we found to be 0.4 pJ) and therefore the power consumption is also negligible compared to the idle power of PCM.

Table 4. Coset Coding Encoder Overheads

Coset Code	Area (μm^2)	Delay (ns)	Avg Energy (pJ)	Max Energy (pJ)
FM-RM(1,3)	1,160	4.09	8.4	10.1
FM-RM(1,7)T	48,344	12.86	56.1	63.4
FM-Parity(72,64)	503	0.84	0.4	0.6

Table 5. Coset Coding Decoder Overheads

Coset Code	Area (μm^2)	Delay (ns)	Avg Energy (pJ)	Max Energy (pJ)
FM-RM(1,3)	344	0.38	0.3	0.3
FM-RM(1,7)T	221	0.59	0.3	0.4
FM-Parity(72,64)	141	0.12	0.1	0.2

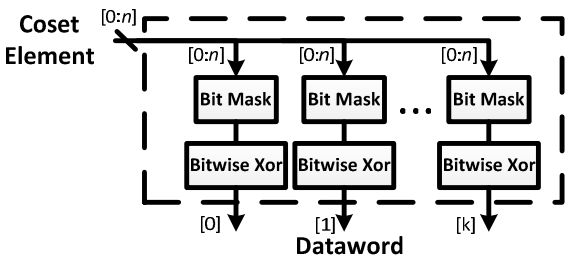


Figure 4. Hardware for decoding

6.3.3 Area

Our worst case area overheads for encoding and decoding are 48344 μm^2 and 344 μm^2 respectively. We believe this to be very small given the overall size of the PCM chips and the size of a typical DIMM.

7. Evaluation

We experimentally evaluated FlipMin to determine how well it can extend memory lifetime and to compare it to prior work. The metric we use to compare memory lifetimes is the number of blocks still usable after a given number of writes.

7.1. Techniques Compared

We compare several different approaches for extending memory lifetime. For all techniques, we consider 64-bit datawords. We evaluate all of the options at the cache line granularity, where a cache line is large enough to hold 8 *n*-bit vectors. For example, the baseline unprotected system has 8 64-bit datawords per line. The schemes compared are:

Bit Flip Reduction Schemes. We evaluate Flip-N-Write [5] on a per-byte granularity—which is equivalent to FlipMin-Parity(72,64)—as well as FlipMin with the codes listed in Table 2.

Error/Erasure Correction Schemes. We compare to schemes that correct errors after they occur. These schemes include traditional Hamming(71,64) ECC, as well as several variants of ECP [23].

Hybrids. We also compare to combinations of bit flip reduction schemes (Flip-N-Write and FlipMin) and erasure tolerance schemes (ECP). We combine FlipMin with both ECP and CEM.

A list of all schemes we evaluated is in Table 6. Note that the three shaded schemes have very similar storage overheads and thus comparisons between them require no normalization to be fair.

7.2. System Model Assumptions

We assume that we can write both transitions (0-to-1 and 1-to-0) to each bit individually without requiring an erase first. Each bit transition has some fixed cost (i.e., incremental wearout) associated with it. This model is known as the Write Efficient Memory (WEM) model [1], and it has been applied before to PCM [3]. We also assume, like prior work [5] [9][26][28][29], that we can

read the memory location before writing it. This is required because choosing the optimal coset element to write relies on knowing the coset element at the location to be written. Fortunately, for the memory technologies we are most interested in, including PCM, reading is far less expensive than writing. Due to the physics of the storage medium, the latency to perform a read and the energy required to perform a read are both far less than for a write. Thus, requiring a read before each write, while not free, has a cost that is dwarfed by the cost of the write itself.

7.3. Modeling Wear-Out

A memory location is usable unless it has more defects than can be tolerated. We consider wearout at the cache block granularity; a block with too many worn-out bits may not "borrow" bits from other blocks. For all schemes compared in this paper, we assume that a block that has failed can be replaced with a spare block (e.g., as is done in hardware in FREE-P [27]). If, instead, a failed block renders the entire page unusable, the results will be qualitatively the same because all of the schemes will suffer in a similar fashion.

At time zero, all bit positions in all memory locations are defect-free and, over time, some of these bit positions wear out and become unusable. Each bit position has a lifetime, measured in the number of writes before it wears out, and we assume that, due to process variability, these lifetimes follow a Gaussian distribution. We assume the mean of this distribution is 10^8 writes, based on published data [30][17]. Assuming a different mean has no impact on the relative lifetimes for different lifetime extension schemes. Changing the mean simply shifts the absolute lifetimes but not the differences between the lifetimes for different schemes. We explore different coefficients of variation (CV equals the mean divided by the standard deviation) in this distribution, in order to determine how the results are impacted by differing amounts of process variability. In particular we looked at a CV of 0.05 which models memory with a tight distribution of lifetimes around the mean, and a CV of 0.2 which models memory cell lifetimes that are farther spread from the mean.

Table 6. Schemes to extend memory lifetime

Scheme	Storage overhead	Bit Flip Reduction	Error/Erasure correction
ECC-Hamming(71,64)	10.9%	0%	8 bits
ECP_6	11.9%	0%	6 bits
ECP_{12}	19.7%	0%	12 bits
ECP_{12}-ideal	0%	0%	12 bits
Flip-N-Write	12.5%	15.8%	0 bits
Flip-N-Write + ECP_6	25.6%	15.8%	6 bits
FM-RM(1,7)T	12.5%	24.5%	0 bits
FM-RM(1,7)T + ECP_6	25.6%	24.5%	6 bits
FM-RM(1,7)T + CEM	24.4%	24.5%	6 bits
FM-RM(1,3)	100%	31.2%	0 bits

7.4. Results for Random Input Streams

In this section, we compare the various lifetime extension schemes when the memory is subjected to a stream of random writes. On average, half of the *dataword* bits flip for each write. In the following figures, we plot the results for both values of the coefficient of variation of the cell lifetime.

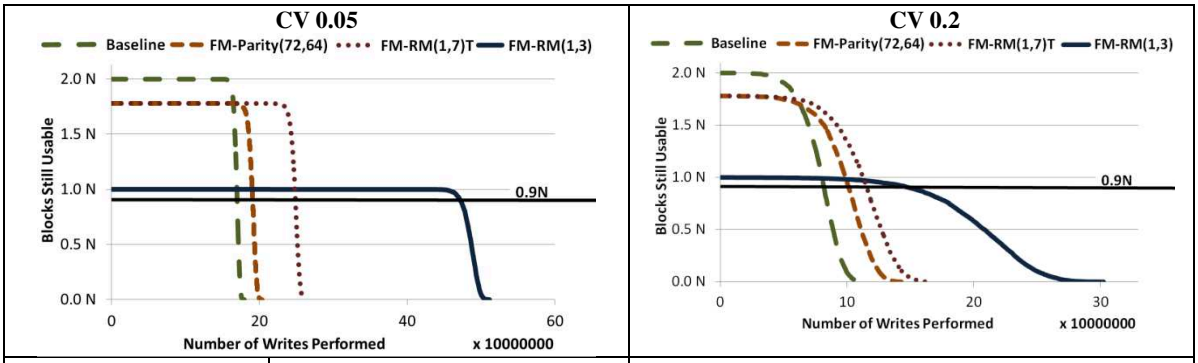
The y-axis in each graph is the number of blocks that are still usable. The number of blocks at time zero is normalized to a value of *N* for the schemes with the highest storage overhead at time zero, which are DoubleMem and FM-RM(1,3) and their 100% overhead. The baseline thus starts at *2N*, and all other schemes start between *N* and *2N*. We say a memory fails when only *0.9N* locations remain. Section 7.4.1 compares schemes for bit flip reduction (i.e., FlipMin and Flip-N-Write), which is the primary focus of this work. Section 7.4.2 compares FlipMin to error/erasure tolerance schemes; even though these schemes have different goals (wearout prevention versus wearout tolerance), they both seek to extend lifetime and thus it is natural to compare them. In Section 7.4.3, we evaluate the intuitively appealing combination of bit flip reduction and erasure tolerance. Finally, Section 7.4.4 looks at a FlipMin code for when much longer memory lifetimes are needed.

7.4.1 Comparison of Bit Flip Reduction Schemes

Figure 5 shows the results when we compare Flip-N-Write and FlipMin. For random inputs, FlipMin with the two RM codes does significantly better than Flip-N-Write at reducing the number of bit flips in the input (recall the data from Table 2) and therefore does better at extending memory lifetime over the baseline. FM-RM(1,7)T achieves a bit flip reduction of 24.5%, which translates into a 46% and 41% improvement over baseline lifetime for CVs of 0.05 and 0.2, respectively. FM-RM(1,3) has a bit flip reduction of 31.2%, which is far better than the other two schemes, but this bit flip reduction comes at the cost of much greater storage overhead. Overall, FM-RM(1,7)T performs better than Flip-N-Write at the same overhead, while FM-RM(1,3) does the best overall, but at higher overheads.

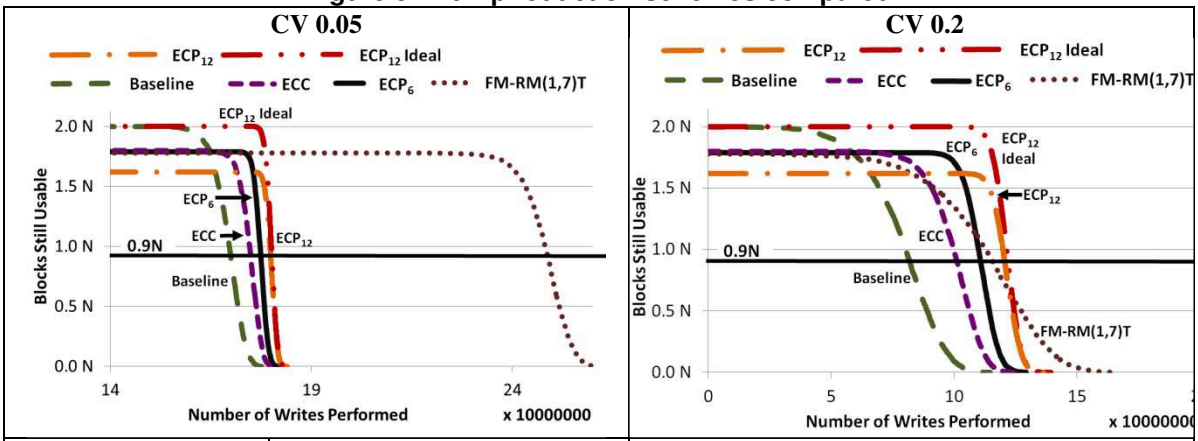
7.4.2 FlipMin vs. Error/Erasure Tolerance

We compared FlipMin to two error/erasure correction schemes: ECC-Hamming(71,64) and ECP. As shown in Figure 6, for a CV of 0.05, ECC-Hamming(71,64) and all variants of ECP (including ECP_{12}-ideal) achieve lifetimes only slightly better than baseline. This is because there are few weak cells, and all cells tend to fail around the same time. FlipMin does well, with FM-RM(1,7)T showing a 46% improvement over baseline.



	CV 0.05		CV 0.2	
Scheme	**Writes Before 0.9N**	**% Improvement Over Baseline**	**Writes Before 0.9N**	**% Improvement Over Baseline**
Baseline	1.70e8	0%	8.20e7	0%
Flip-N-Write	1.91e8	12%	1.01e8	23%
FM-RM(1,7)T	2.49e8	46%	1.16e8	41%
FM-RM(1,3)	4.72e8	178%	1.49e8	82%

Figure 5. Bit flip reduction schemes compared



	CV 0.05		CV 0.2	
Scheme	**Writes Before 0.9N**	**% Improvement Over Baseline**	**Writes Before 0.9N**	**% Improvement Over Baseline**
Baseline	1.70e8	0%	8.20e7	0%
ECC-Hamming(71,64)	1.75e8	3%	1.01e8	23%
ECP_6	1.78e8	5%	1.11e8	35%
ECP_{12}	1.80e8	6%	1.20e8	46%
ECP_{12}-ideal	1.80e8	6%	1.21e8	48%
FM-RM(1,7)T	2.49e8	46%	1.16e8	41%

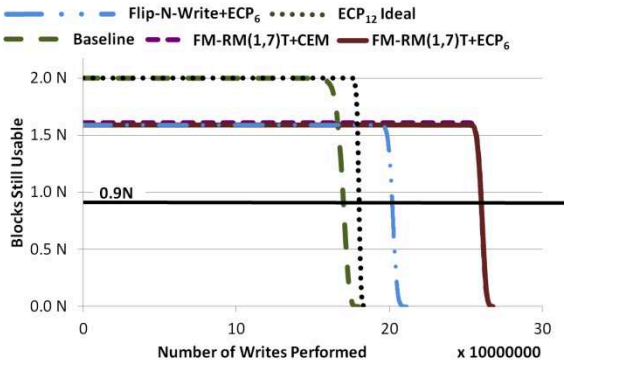
Figure 6. FlipMin compared to error/erasure tolerance schemes

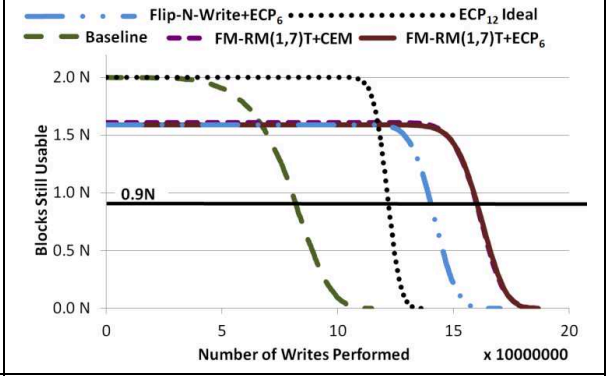
If the cell lifetime distribution is grouped together tightly, FlipMin is much better than error/erasure correction schemes. However, for a large CV of 0.2, ECP is far more helpful than before, because it can tolerate the weak cells that are outliers in the lifetime distribution. This is the scenario that ECP targets. FM-RM(1,7)T is still slightly more effective than ECP_6 yet slightly less effective than ECP_{12}.

7.4.3 Bit Flip Reduction + Erasure Tolerance

The previous two sections have shown how well FlipMin can do if the lifetimes of the memory cells are relatively uniform. We have also shown that for higher variations in cell lifetime, we probably want some sort of erasure tolerance to deal with early cell failures. To get both of these benefits, we combined bit flip reduction with erasure tolerance. The properties of the hybrid schemes we consider are listed in Table 6. The results are shown in Figure 7. If the cell lifetime CV is low, adding erasure tolerance is not that helpful.

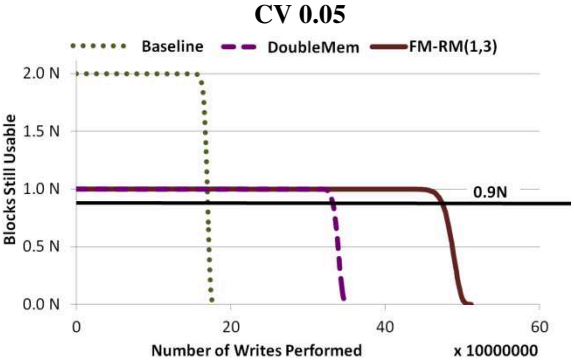
However, at a high CV of 0.2, the combination of FM-RM(1,7)T and erasure tolerance (either CEM or ECP6) results in a lifetime gain of 95%, which is far greater than either FM-RM(1,7)T or ECP_{12} alone. This result highlights the synergistic effects of FlipMin and

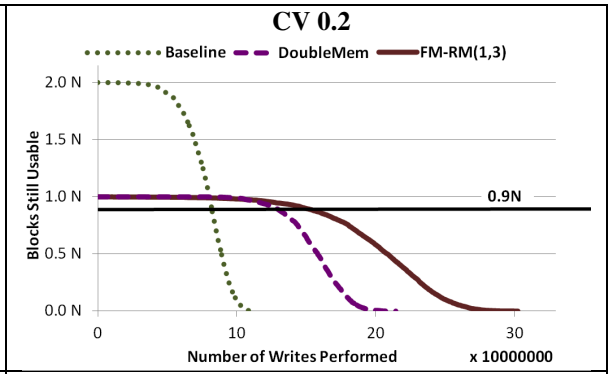




Scheme	CV 0.05		CV 0.2	
	Writes Before 0.9N	% Improvement Over Baseline	Writes Before 0.9N	% Improvement Over Baseline
Baseline	1.70e8	0%	8.20e7	0%
ECP_{12}-ideal	1.80e8	6%	1.21e8	48%
Flip-N-Write + ECP_6	2.02e8	19%	1.40e8	71%
FM-RM(1,7)T + ECP_6	2.60e8	53%	1.60e8	95%
FM-RM(1,7)T + CEM	2.60e8	53%	1.60e8	95%

Figure 7. Bit flip reduction + erasure tolerance compared to error/erasure tolerance alone





Scheme	CV 0.05		CV 0.2	
	Writes Before 0.9N	% Improvement Over Baseline	Writes Before 0.9N	% Improvement Over Baseline
Baseline	1.70e8	0%	8.20e7	0%
DoubleMem	3.31e8	95%	1.28e8	56%
FM-RM(1,3)	4.72e8	178%	1.49e8	82%

Figure 8. FM-RM(1,3) coset coding compared to DoubleMem

erasure tolerance at high CVs: if we take the lifetime gains for both ECP_6 and FM-RM(1,7)T individually from Figure 6, the total is 35% + 41% = 76%, which is less than the 95% of the combined scheme. Combining Flip-N-Write with ECP_6 does strictly worse than FlipMin with ECP_6.

7.4.4 FlipMin for Longer Memory Lifetimes

We have shown that FM-RM(1,7)T provides superior lifetime gains when compared to existing erasure tolerance schemes at the same area overhead. We have also shown that we can combine FlipMin and erasure tolerance to combat high memory cell lifetime variation. There may be cases where even larger lifetime gains are required. For these cases, FM-RM(1,3) provides substantial lifetime gains, but at the cost of 100% area overhead at time zero. Since we could just as easily use two memory locations with the same overhead, we use that as our point of comparison. Specifically we use a technique we call DoubleMem that we described in Section 2.4. We ignore the overhead required to track which lines are being used by DoubleMem, thus giving it an unrealistic advantage.

As shown in Figure 8, at both CV points, FM-RM(1,3) easily outperforms DoubleMem. FM-RM(1,3) gives 178% additional lifetime at a CV of 0.05 while DoubleMem gives only 95% lifetime gains over baseline. For a CV of 0.2, lifetime gains are 82% for FM-RM(1,3) and 56% for DoubleMem due to the

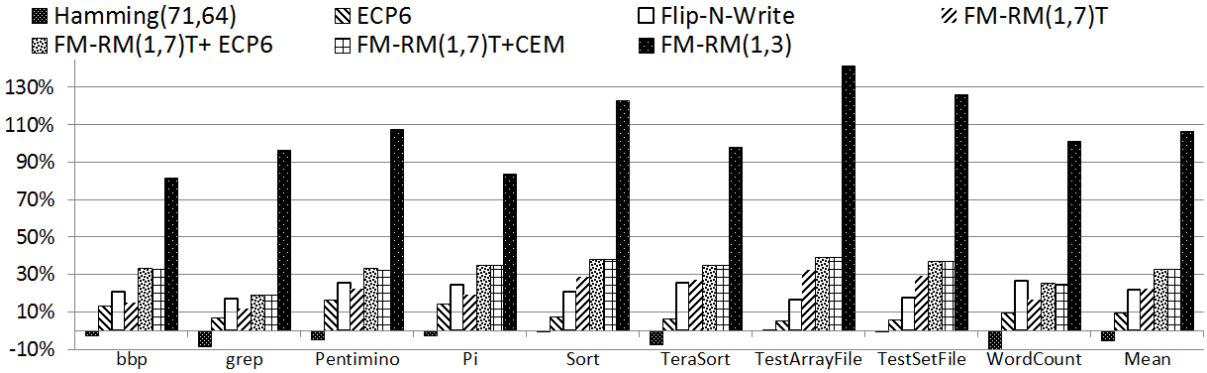


Figure 9. Lifetime extension schemes compared using Hadoop inputs. CV 0.05

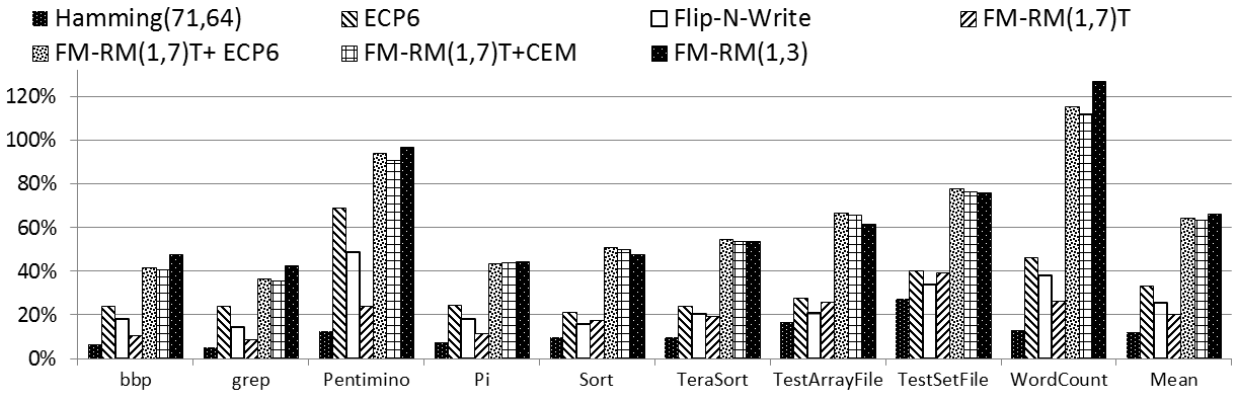


Figure 10. Lifetime extension schemes compared using Hadoop inputs. CV 0.2

increased number of weak cells. The results show that adding memory cells can increase lifetime and that it is important *how* we use those extra memory cells, as shown by FM-RM(1,3)'s advantage over DoubleMem. Because DoubleMem does worse than FM-RM(1,3) for all benchmarks we do not consider it further.

7.5. Results for Benchmarks

We now show how FlipMin performs when the system runs benchmarks. We used the Gem5 simulator [2] to simulate a single core configured as shown in Table 7. For benchmarks, we used the Hadoop benchmarks that come with the Hadoop distribution. One challenge was obtaining a large number of writes to a large number of lines in a reasonable amount of time. We solved this problem by dumping traces of writes to every line, then projecting these traces onto one target page of memory with lines numbered from *1* to *B*. The projection is done by mapping line *X* to line *X* modulus *B*. The trace of writes to the target page's first line is the concatenation of the traces for lines *1*, *B+1*, *2B+1*, etc.

Our results for Hadoop are in Figure 9 and 10. The y-axis is the gain in lifetime, where lifetime is measured as the number of writes until *0.9N* locations remain. All FlipMin schemes do significantly better than both baseline and the error correcting schemes, including ECP_6, at a CV of 0.05. On average, ECP_6 produces a 9.2% improvement in lifetime over baseline, while FM-RM(1,7)T produces a 22% lifetime improvement over baseline. For the higher CV of 0.2, ECP_6 alone does well with a 33% lifetime gain over baseline, but when combined with FlipMin it does even better with an overall lifetime improvement over baseline of 64%. The larger overhead code of FM-RM(1,3) does very well at all CVs with a lifetime gain over baseline of 106% for a CV of 0.05 and 66% for a CV of 0.2.

CPU	x86-64 in-order core at 2.0GHz
L1 D-Cache	64kB, 2-way associative, 64B Line Size
L1 I-Cache	32kB, 2-way associative, 64B Line Size
L2 Cache	2MB, 8-way associative, 64B Line Size
Memory	128MB, 2GB swap

Table 7. System configuration

On some, but not all benchmarks, Flip-N-Write outperforms FM-RM(1,7)T, even though FM-RM(1,7)T outperforms Flip-N-Write on random inputs. This discrepancy highlights how lifetime results depend on the inputs. In particular, if fewer bits flip per dataword (e.g., in benchmarks like WordCount), then shorter codes, like Flip-N-Write and RM(1,3), do better than longer codes like RM(1,7)T. This discrepancy also highlights the tunability of FlipMin, by showing the range of results achievable with a variety of codes.

8. Conclusions

We have shown that coset coding enables us to optimize writing memory structures. In particular, we

have shown how to use coset coding to avoid wearout—by reducing bit flips—and to then tolerate the bits that do eventually wear out. Furthermore, we have by no means exhausted the possible coset coding techniques that we can use. There are more possible codes and metrics, which we hope to explore in future work.

Acknowledgments

This material is based on work supported by the National Science Foundation under grant CCF-111-5367.

References

[1] R. Ahlswede and Z. Zhang, "Coding for Write-Efficient Memory," *Information and Computation*, vol. 83, no. 1, pp. 80–97, Oct. 1989.

[2] N. Binkert et al., "The Gem5 Simulator," *Computer Architecture News*, 39(1), Aug. 2011.

[3] G. W. Burr et al., "Phase change memory technology," *Journal of Vacuum Science & Technology B: Microelectronics and Nanometer Structures*, 28(2), 2010.

[4] A. R. Calderbank and N. J. A. Sloane, "New Trellis Codes Based on Lattices and Cosets," *IEEE Trans. Info. Theory*, vol. 33, 1987.

[5] S. Cho and H. Lee, "Flip-N-Write: A Simple Deterministic Technique to Improve PRAM Write Performance, Energy and Endurance," in *Proceedings of the 42nd Annual Int'l Symp. on Microarchitecture*, 2009.

[6] G. D. Forney, "Coset Codes. I. Introduction and Geometrical Classification," *IEEE Trans. Information Theory,* 34(5), Sep. 1988.

[7] G. D. Forney, "Coset Codes. II. Binary Lattices and Related Codes," *IEEE Trans. Information Theory,* 34(5), Sep. 1988.

[8] G. D. Forney, "Trellis Shaping," *Information Theory, IEEE Transactions on*, vol. 38, no. 2, Mar. 1992.

[9] E. Ipek et al., "Dynamically Replicated Memory: Building Reliable Systems from Nanoscale Resistive Memories," in *Proc. of the Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, 2010.

[10] A. Jagmohan, M. Franceschini, and L. Lastras, "Write Amplification Reduction in NAND Flash Through Multi-Write Coding," in *IEEE 26th Symp on Mass Storage Systems and Technologies (MSST)*, 2010.

[11] A. Jagmohan et al., "Coding for Multilevel Heterogeneous Memories," in *IEEE Int'l Conference on Communications*, 2010.

[12] Y. Joo, D. Niu, X. Dong, G. Sun, N. Chang, and Y. Xie, "Energy- and Endurance-Aware Design of Phase Change Memory Caches," in *Proc. of Conf. on Design, Automation & Test in Europe*, 2010.

[13] G. Keshet, Y. Steinberg, and N. Merhav, "Channel Coding in the Presence of Side Information," *Found. Trends Commun. Inf. Theory*, vol. 4, no. 6, Jun. 2008.

[14] L. A. Lastras-Montaño et al., "On the Lifetime of Multilevel Memories," in *Proc. of the IEEE Int'l Symposium on Information Theory - Volume 2*, 2009.

[15] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting Phase Change Memory as a Scalable DRAM Alternative," in *Proceedings of the 36th Int'l Symposium on Computer Architecture*, 2009.

[16] R. Melhem, R. Maddah, and S. Cho, "RDIS: A Recursively Defined Invertible Set Scheme to Tolerate Multiple Stuck-At Faults in Resistive Memory," in *42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*.

[17] Micron Technology, Inc., "P8P Parallel Phase Change Memory (PCM) NP8P128A13B1760E." Micron, 2005.

[18] Nangate Development Team, "Nangate 45nm Open Cell Library." 2012.

[19] M. K. Qureshi, "Pay-As-You-Go: Low-Overhead Hard-Error Correction for Phase Change Memories," in *Proceedings of the 44th Annual International Symposium on Microarchitecture*, 2011.

[20] M. K. Qureshi et al., "Enhancing Lifetime and Security of PCM-Based Main Memory with Start-Gap Wear Leveling," in *Proceedings of the 42nd Annual Int'l Symposium on Microarchitecture*, 2009.

[21] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable High Performance Main Memory System Using Phase-Change Memory Technology," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, 2009.

[22] I. Reed, "A Class of Multiple-Error-Correcting Codes and the Decoding Scheme," *IRE Professional Group on Information Theory*, vol. 4, no. 4, pp. 38–49, Sep. 1954.

[23] S. Schechter, G. H. Loh, K. Strauss, and D. Burger, "Use ECP, Not ECC, for Hard Failures in Resistive Memories," in *Proc. of the 37th Annual Int'l Symposium on Computer Architecture*, 2010.

[24] N. H. Seong, D. H. Woo, and H.-H. S. Lee, "Security Refresh: Prevent Malicious Wear-Out and Increase Durability for Phase-Change Memory with Dynamically Randomized Address Mapping," in *Proc. 37th Int'l Symp. on Computer Architecture*, 2010.

[25] N. H. Seong et al., "SAFER: Stuck-At-Fault Error Recovery for Memories," in *43rd Annual International Symposium on Microarchitecture*, 2010.

[26] B.-D. Yang et al., "A Low Power Phase-Change Random Access Memory using a Data-Comparison Write Scheme," in *Proc. of the IEEE International Symposium on Circuits and Systems*, 2007.

[27] D. H. Yoon et al., "FREE-p: Protecting Non-Volatile Memory Against Both Hard and Soft Errors," in *Proc. of the International Symposium on High-Performance Computer Architecture*, 2011.

[28] W. Zhang and T. Li, "Characterizing and Mitigating the Impact of Process Variations on Phase Change Based Memory Systems," in *Proc. of the 42nd Annua Int'l Symposium on Microarchitecture*.

[29] P. Zhou et al., "A Durable and Energy Efficient Main Memory Using Phase Change Memory Technology," in *Proc. 36th Int'l Symp. on Computer Architecture*, 2009.

[30] *International Technology Roadmap for Semiconductors, 2007 Edition, Process Integration, Devices, and Structures*. 2007.