GreenFlag: Protecting 3D-Racetrack Memory from Shift Errors

Georgios Mappouras
Electrical & Computer Engineering
Duke University
Durham, USA
gm118@duke.edu

Alireza Vahid
Electrical Engineering
University of Colorado Denver
Denver, USA
alireza.vahid@ucdenver.edu

Robert Calderbank
Electrical & Computer Engineering
Duke University
Durham, USA
robert.calderbank@duke.edu

Daniel J. Sorin
Electrical & Computer Engineering
Duke University
Durham, USA
sorin@ee.duke.edu

***Abstract*—Racetrack memory is an exciting emerging memory technology with the potential to offer far greater capacity and performance than other non-volatile memories. Racetrack memory has an unusual error model, though, which precludes the use of the typical error coding techniques used by architects. In this paper, we introduce GreenFlag, a coding scheme that combines a new construction for Varshamov-Tenegolts codes with specially crafted delimiter bits that are placed between each codeword. GreenFlag is the first coding scheme that is compatible with 3D racetrack, which has the benefit of very high density but the limitation of a single read/write port per track. Based on our implementation of encoding/decoding hardware, we analyze the trade-offs between latency, code length, and code rate; we then use this analysis to evaluate the viability of racetrack at each level of the memory hierarchy.**

Keywords—Racetrack Memory; Coding; Fault Tolerance; Shift Errors

I. Introduction

Many new non-volatile memory technologies are vying to replace conventional memory technologies—such as SRAM, DRAM, and Flash—and *racetrack memory* [1, 2, 3] has the potential to provide the best storage density and performance of any of these contenders. In Table I, we compare racetrack memory to two of the other emerging non-volatile memories as well as to SRAM. Competing technologies, such as phase change memory (PCM) and magneto-resistive random-access memory (MRAM), are at a disadvantage in terms of density and performance [4, 5, 6, 7, 8, 9, 10]. SRAM is somewhat faster than racetrack memory, but it is volatile and two orders of magnitude less dense. These quantitative advantages—combined with racetrack's compatibility with standard fabrication processes and promising results in research labs [2, 5, 8, 10, 11]—motivate us to explore the potential for racetrack memory to be used in computer architectures.

As we explain in more detail in Section II, racetrack's unique design provides both the advantages mentioned above but also a new reliability challenge. Racetrack memory stores bits on a large number of nanowire tracks that can each be accessed one bit at a time with a fixed port; the bits are shifted along the fixed track such that the desired bit position is over the port. This design enables excellent storage density and short access latencies, but it is unfortunately susceptible to an error model that is unfamiliar to architects: shift errors. Shift errors include both deletions and insertions [12, 13]. A deletion occurs when the track is shifted more than expected and thus one (or more) bits are skipped, i.e., the memory is over-shifted. An insertion occurs when the track is shifted less than expected and the bit under the port does not change, and we read the same bit twice (or even more), i.e., the memory is under-shifted.

TABLE I. COMPARING MEMORY TECHNOLOGIES

	PCM	MRAM	SRAM	Racetrack
Volatile	No	No	Yes	No
Density (F^2)	4-16	20-60	140	1-2
Read (ns)	10-50	10-35	1-10	3-10
Write (ns)	50-500	10-90	1-10	10-20

The goal for architects is to tolerate shift errors without sacrificing too much of the latency and density benefits provided by racetrack memory. Error coding—for any memory technology—exhibits a fundamental tension between *code length* (i.e., how many bits are in each codeword), *code rate* (i.e., the ratio of the data bits to the sum of the data bits and the extra bits required for the code), and latency. Achieving our goal for racetrack memory is complicated by both its unusual error model (shift errors) and its bit-serial access nature, because the only ways to improve latency are to read from multiple tracks in parallel or use shorter codewords.

Consider a grid of racetrack memory, in which each track is a horizontal row (even if the track itself is 3D, as discussed later), and each column is the collection of bits at the same bit position in each track. Assume we want to be able to read (or write) C bits at a time. For the best latency, we would prefer to read one bit from each of C tracks—achieving a parallelism of C—and thus achieve single-cycle accesses. In theory, we could do this by coding "vertically", i.e., encoding information on a per-column basis. Unfortunately, as we show in Section III, known vertical coding schemes do not suffice. Vertical coding may be possible, but such a solution does not exist today.

If we cannot code vertically, we must code "horizontally" by encoding information in a group of C bits on a given track. Each track would be encoded independently. We show in Section III that commonly used codes like Hamming cannot handle shift errors, so we have developed a new coding

978-1-7281-0057-9/19/$31.00 ©2019 IEEE
DOI 10.1109/DSN.2019.00016

technique, called GreenFlag[1], that composes Varshamov-Tenegolts (VT) codes [14] with specially crafted delimiter bits that detect and correct shift errors. The architectural trade-off is that longer GreenFlag codes achieve a better rate but incur a longer read latency. (The analysis for bandwidth is more subtle but bandwidth is far less sensitive than latency to the choice of code.) Assume that codeword length is C bits and we wish to access B bits ($B>C$). The best parallelism we can achieve is to read C bits on each of B/C tracks, thus achieving a parallelism of B/C. As C increases, the trade-off is that parallelism decreases (and thus latency increases) and rate increases. The exact results for latency and bandwidth depend on the hardware, so we have implemented and evaluated the circuitry for encoding datawords and decoding codewords.

There is one clever but limited exception to the above analysis, which is HiFi [12]. HiFi is a horizontal "code" that can detect and correct errors at the granularity of a single bit. In the terms of horizontal coding above, it has the ideal C=1 and parallelism of B. However, HiFi requires multiple ports on each track, and that is only possible with 2-dimensional (2D) tracks. Because 3D tracks can offer vastly greater density than 2D tracks [5, 11, 15, 16], we do not consider HiFi or any other possible scheme that is constrained to 2D tracks.

The architectural viability of racetrack depends on the possible trade-offs between code length, rate, latency, and bandwidth, so we analyzed these trade-offs for racetrack with GreenFlag at each level of the memory hierarchy. Our goal is to determine the viability of Racetrack with effective error tolerance, not to promote Racetrack as necessarily the best option, and our analysis is thus more of a limit study than a cycle-accurate comparison against other schemes. For a given level of the memory hierarchy, the viability of racetrack memory is determined by B/C. Specifically, assume a given level of cache requires an access latency to B bits that is no longer than a specified amount of time (e.g., 20ns for an L3 cache). That latency determines the required access parallelism B/C and, because B is fixed, determines the codeword length C. In turn, C determines the rate of the code, and we explore this relationship between C and rate for GreenFlag codes. If the rate is too low, racetrack might be considered unattractive compared to existing memory technologies. For example, we show that the best rate we can achieve for a LLC cache with a 50ns access latency is 0.125. It is unlikely that racetrack at this rate is preferable to simply using SRAM.

We make the following contributions in this paper:

- We present the GreenFlag coding scheme that combines a novel construction of VT codes with specially crafted delimiter bits to efficiently detect and correct shift errors.
- We implement and evaluate the GreenFlag hardware circuitry required to encode datawords and decode codewords.
- We present the first analysis of the viability of racetrack memory—based on the trade-off between code length, code rate, latency, and bandwidth—at each level of the memory hierarchy.

II. Racetrack Memory

This section introduces the physical model of racetrack memory and describes its error model.

A. *Racetrack Background*

Racetrack memory stores data in tape-like tracks. Each track stores data bits in magnetic domains and neighboring domains are separated by a domain wall. All read/write ports and the physical substrate are fixed in position, and as spin-coherent electric current is passed through a track, its domains shift by the magnetic read/write port positioned near the track.

Although these tracks can be manufactured in two or three dimensions—sometimes referred to as horizontal and vertical racetrack memory, respectively—the three-dimensional structure is preferred as it can offer dramatically greater density [5, 11, 16]. With 3D racetrack, the tracks are in a U-shaped geometry in three dimensions, and the read/write ports are fixed in position at the bottom of this structure as illustrated in Fig. 1. The 3D structure of racetrack memory limits the feasible number of read/write ports per track to one.

Because of the huge density benefits of 3D racetrack memory, we consider only 3D tracks and, thus, only coding schemes that can be implemented with one read/write port per track. However, for simplicity, when we illustrate tracks to facilitate the description of our code in the following sections, we use a 2D schematic representation.

B. *Error Model*

Our error model includes single shift and double shift errors[2]. We do not consider triple or higher shift errors as prior work [12] has shown them to have negligible probability. Similar to prior work [12], we do not include bit-flip errors, as we have discovered no data on this phenomenon in the literature. We do not claim that bit-flip errors are impossible in racetrack memory; if future evidence of them appears, we would need to extend our work here to address them.

To explain shift errors, we use an example in which a number of bits is stored on a track as in Fig. 2(a). To read

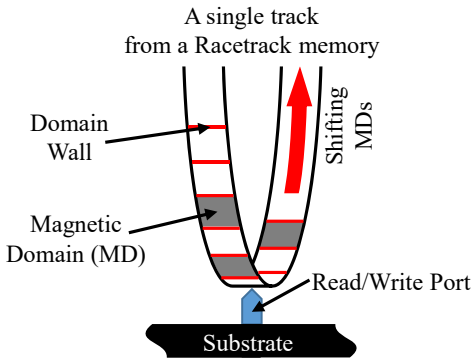

Fig. 1. 3D (vertical) racetrack

[1]In car racing, a green flag indicates good track conditions.

[2]A single shift error occurs when a single shift operation deletes or inserts a single bit. A double shift error occurs when a *single* shift operation deletes or inserts two bits.

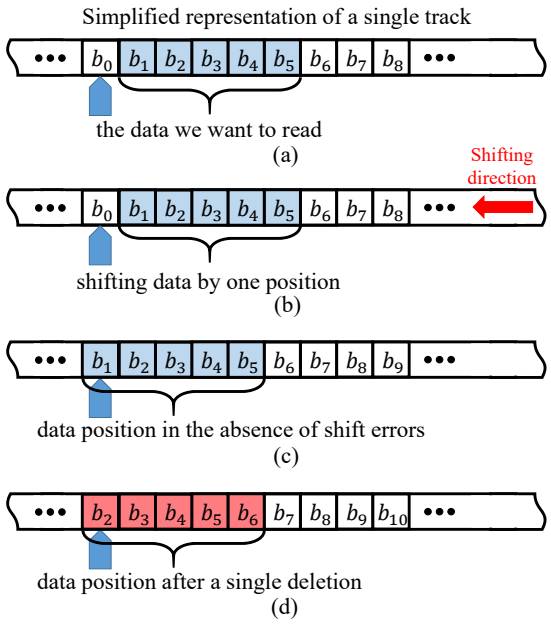

Fig. 2. Error-free shifting and shift errors

(write) the stored bits, we need to perform a sequence of consecutive shift and read (write) operations. A shift is the injection of current, shown in Fig. 2(b), so as to place the next bit in racetrack memory under the read/write port.

Suppose we want to read bit b_1 and currently bit b_0 is positioned under the read/write port. Ideally, we inject the correct amount of current to perform a single shift operation and b_1 is placed under the read/write port as in Fig. 2(c). If the injected current is larger than expected, then we skip the domain that contains b_1 and b_2 is wrongly positioned under the read/write port as in Fig. 2(d). We refer to this error as a *deletion* error since b_1 is skipped (i.e., a bit is deleted). On the other hand, if the injected current is smaller than expected, the port's position does not change and we might read the same bit twice. We refer to this error as an *insertion* error (i.e., an extra bit is inserted).

We should emphasize that, regardless of whether a shift error occurs, the memory controller always provides the desired number of bits, say n. A shift error just affects which n bits are provided. For instance, if a deletion happens as in Fig. 2(d), then we receive the first $n = 5$ bits excluding b_1, which are b_2, b_3, b_4, b_5, b_6.

We observe that a single deletion (or insertion) can cause as many as n bit errors, which greatly complicates error coding, as we show later.

III. Error Coding for Racetrack

Coding for racetrack is different than for typical memory technologies because of its bit-serial nature (which affects performance) and its susceptibility to shift errors (which makes many standard codes ineffective).

We now discuss two broad approaches to coding for racetrack—vertical (across tracks) and horizontal (within a track)—and show why commonly used coding techniques do not work.

To clarify the explanation, we use the well-known Hamming code as a running example. Hamming codes add parity bits to datawords to form codewords, and they are parameterized by the bit lengths of the datawords and codewords. With no loss of generality, we assume a Hamming(8,4) code that encodes 4-bit datawords as 8-bit codewords. The code provides SECDED protection.

A. *Vertical Coding*

Ideally, for performance, we would employ a vertical code that uses domains across multiple tracks to store a codeword, as illustrated in Fig. 3. We would use one bit from each of C different tracks to store the C bits of the codeword. With such a code, we could read all C bits in parallel with a single shift and read operation on each track.

In Fig. 3(a) we illustrate an 8-bit codeword c =10101010 from the Hamming (8,4) code striped vertically across $C = 8$ tracks. Fig. 3(b) shows the state of these 8 tracks after a single deletion in the first track changes c to c'= 00101010.

Only one bit position is affected and thus a SECDED Hamming code like Hamming(8,4) can correctly recover c. However, the code cannot fix the still-erroneous position of the domains on that track with respect to the read/write port. Note that, because the code cannot differentiate between deletions and insertions, the codeword bits that are striped across tracks cannot be correctly aligned. Once the position of the domains becomes incorrect with respect to the port, it remains incorrect, which can lead to further errors on that track. Worse, a subsequent error on another track can now lead to a multiple-error scenario that cannot be corrected by a SECDED code. Thus, a shift error on one track can easily lead to a situation in which the data on all 8 tracks are lost.

We are unaware of any existing coding scheme that would overcome the problem we have just described. We do not claim that such a code is impossible, but to our knowledge, it has not yet been invented.

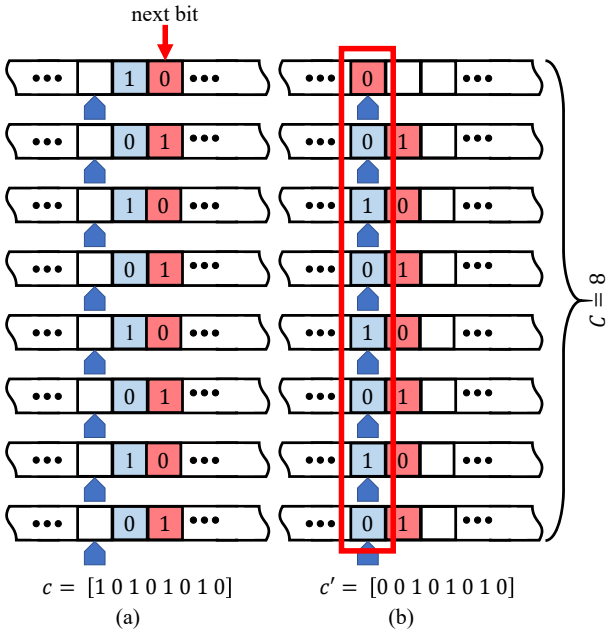

Fig. 3. Hamming code – vertical implementation

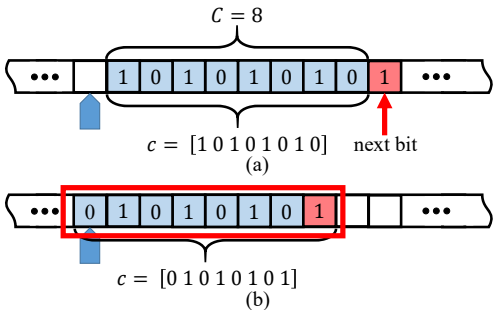

Fig. 4. Hamming code – horizontal implementation

B. *Horizontal Coding*

Given that vertical coding is not possible, we are left with horizontal coding, in which each *C*-bit codeword is written on the domains of a single track, as illustrated in Fig. 4. The main drawback of horizontal coding is that we need to perform *C* read and *C* shift operations to read all the domains (bits) of a single codeword. That means that latency depends on *C*.

To explain why standard codes do not suffice for handling shift errors, we use an example.[3]

Our Hamming(8,4) code has a generate matrix G shown below.

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

Assume that we want to store the dataword $d = [1\ 0\ 1\ 0]$ on a track. The dataword is first encoded to the codeword *c* and then written to the track as shown in Fig. 4(a).

$$c = d * G = [1\ 0\ 1\ 0\ 1\ 0\ 1\ 0]$$

Assume that a single deletion happens during the first shift operation while we are trying to read *c*. In Fig. 4(b) we present the state of the track after the deletion error converts *c* into c'.

$$c' = [\,0\ 1\ 0\ 1\ 0\ 1\ 0\ 1]$$

We observe that c' differs from c in every bit position, which overwhelms the capability of a SECDED code. Nevertheless, it is instructive to see the resulting dataword d' that is produced by our code:

$$c' = d' * G = [0\ 1\ 0\ 1\ 0\ 1\ 0\ 1\,]$$
$$d' = [0\ 1\ 0\ 1]$$

We observe that d' differs from the correct dataword *d* in every bit position, despite experiencing only a single shift error. Through this example it is easy to see that, even if we had used stronger codes (e.g., DECTED), we still could not address shift errors. Additionally, even if we could detect and correct the bit-value errors, the position of the read/write port with respect to the domains would still be wrong. Thus, any future read/shift operations would suffer additional bit-value errors.

C. *Conclusion*

Based on our observations thus far, we have the following three goals for GreenFlag:

- GreenFlag must be a horizontal code.
- GreenFlag cannot be a typical code, like Hamming (or Reed-Solomon, CRC, etc.).
- GreenFlag should be compatible with 3D racetrack and its limitation of one read/write port per track.

IV. Varshamov-Tenengolts Codes

GreenFlag coding is based on Varshamov-Tenengolts (VT) codes [14], which are part of a family of graph-based codes. VT codes are constituents of communication systems where deletions are common, but we are unaware of any prior use in computer systems. We note that graph-based codes have been proposed to correct deletion errors in communication systems [17].

A. *Graph Codes*

To explain VT codes, we first present graph codes in general, their key idea, and a simple example. We then formally define VT codes.

We denote binary strings with boldface letter (e.g., $\boldsymbol{x}$), and bits in these strings are denoted by lower case letters (e.g., x_1). To label different binary strings, we use different boldface letters, e.g., $\boldsymbol{x}$ and $\boldsymbol{y}$.

Key idea: Consider two binary strings $\boldsymbol{x} = x_1, x_2, \dots, x_n$ and $\boldsymbol{y} = y_1, y_2, \dots, y_n$. In order to be able to use these two strings to store different values in racetrack memory, we need to be able to distinguish them even after a single deletion. In other words, if x_i and y_j are deleted for $i, j \in \{1,2, \dots, n\}$, then the resulting binary strings should not be identical. If two binary strings $\boldsymbol{x}$ and $\boldsymbol{y}$ can be confused with a single deletion, we refer to them as *conflicting*; if they cannot be confused with a single deletion, we refer to them as *non-conflicting*. Consider the following set of binary strings.

$$\left\{ \begin{matrix} 00000 & 00011 & 01110 & 10101 & 11000 & 11111 \\ \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\ \boldsymbol{c_0} & \boldsymbol{c_1} & \boldsymbol{c_2} & \boldsymbol{c_3} & \boldsymbol{c_4} & \boldsymbol{c_5} \end{matrix} \right\}$$

These six strings/codewords are pairwise non-conflicting. Let D_{c_j} denote the set of all strings of length 4 that result from a single deletion in c_j. In this example, $D_{c_0} = \{0000\}$ while $D_{c_3} = \{0101, 1101, 1001, 1011, 1010\}$. Thus, if we read out "0101" we know that it refers to codeword $\boldsymbol{c_3} = 10101$.

Formal definition: Consider all 2^n binary strings of length n. We create a conflict graph $N = (V, E)$, where V is the node set and $E \subseteq V \times V$ is the edge set. Each node in this graph corresponds to a unique binary string of length n ($|V| = 2^n$). For simplicity, we label the nodes with numbers $0, 1, 2, \dots, 2^n - 1$ and we let node i correspond to the binary expansion (of length n) of i that we denote by $\boldsymbol{b_i}$.

Now, $(i, j) \in E$ if and only if $\boldsymbol{b_i}$ and $\boldsymbol{b_j}$ are conflicting. Fig. 5 depicts a conflict graph N for n=3. In this graph, a set of non-conflicting strings corresponds to an *independent set*; the maximum number of strings of length n corresponds to the size of the *maximum independent set*. This problem is

[3] HiFi [12] also illustrates how typical codes fail to tolerate shift errors. We provide this explanation here for completeness.

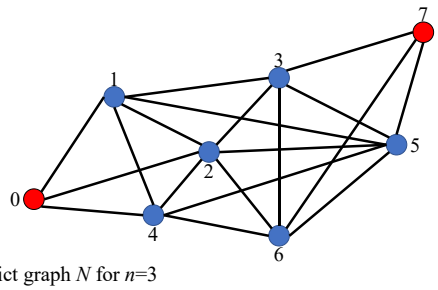

Fig. 5. Conflict graph *N* for *n*=3

well-studied in graph theory. For a given n, we denote the maximum independent set by I_n. We use the binary strings corresponding to the nodes in the maximum independent set as our codewords. We note that the maximum independent set of a graph is not necessarily unique. For the graph in Fig. 5, the maximum independent set is of size two and includes nodes 0 and 7.

Although we have demonstrated only how a graph code works for a deletion, the explanation for insertions is analogous, and a graph code that can tolerate a single deletion can also tolerate a single insertion.

There are two major challenges with using graph codes that we will address when we present GreenFlag in Section V. First, encoding and decoding require the use of look-up tables, and those tables can have significant space and latency overheads. Second, if decoding reveals a shift error, we can only correct it if we know whether it was a deletion or an insertion. That problem has not arisen for communication systems that consider only deletion or insertion errors but not a mixture of both [18, 19, 20, 21, 22, 23, 24].

B. *Varshamov-Tenengolts Codes*

We have already shown that it is possible to use graph codes to tolerate shift errors, but there are two challenges that must be overcome. One of those challenges—requiring lookup tables for encoding and decoding, which does not scale to long codes—has been addressed by a special family of graph codes called Varshamov-Tenengolts (VT) codes.

VT codes impose an additional constraint on the graph-based codes in order to enable faster and more efficient encoding and decoding.

Formal definition: a VT(n) code consists of all binary strings $\boldsymbol{c} = (c_1, c_2, \dots, c_n)$ that satisfy:

$$\sum_{i=1}^{n} i c_i \equiv 0 \pmod{n+1} \qquad \text{Equation 1}$$

where the sum is evaluated as an ordinary rational integer. We refer to Equation 1 as the checksum.

The encoding and decoding algorithms of VT codes depend on their construction (i.e., how parity bits are generated and at which positions they appear in the codeword). There are several ways to construct VT codes, and in Section V we provide a novel construction method that allows for efficient encoding and decoding.

Detecting a shift error with a VT code is as simple as calculating the checksum. If the checksum is zero, then there was no shift error. Otherwise a shift error occurred.

While error detection is simple, error correction is more complicated. As with graph codes in general, VT codes can only correct a shift error if we can first categorize it as a deletion or insertion. Furthermore, we must be able to detect "silent" shift errors, in which the shift error does not corrupt the currently read codeword.

Consider the example in Fig. 6. Assume we are reading codewords from the example code provided in Section IV.A for $n = 5$. In Fig. 6(a) we show the state of the track while we are attempting to read the codeword 00011, which is followed by the codeword 11111. When we attempt to read the last bit of the first codeword, a deletion occurs as shown in Fig. 6(b). We still correctly read 00011 and the shift error is not detected (i.e., the shift error is silent at this point). While this situation may not seem problematic because there are no bit-value errors, the relative position of the read/write port is still misplaced. If an additional shift error occurs while reading in the next codeword it may result in an undetected error.

The problem arises due to the inability to distinguish the boundaries between codewords. Prior work [25, 26] has used predetermined patterns (sometimes called commas) at the end of each codeword to separate consecutive codewords and address this boundary problem. We call these patterns delimiters and they have been used in prior work to detect deletions or insertions but not a mixture of both. In Section V, we present a new set of delimiters we have crafted to distinguish deletions from insertions and that even allow us to detect double deletion and insertion errors.

V. GreenFlag Coding

In this section, we present GreenFlag coding. To create GreenFlag, we have integrated a VT-based horizontal code with specially crafted delimiter bits that allows us to detect and correct not only single but also double shift errors.

We first explain how we construct VT codes for GreenFlag and how we encode and decode in the error-free case. Then we introduce different delimiter options to help us categorize detected shift errors as deletions or insertions. Finally, we describe how GreenFlag corrects shift errors by both fixing erroneous bits and moving the correct racetrack domain under the read/write port.

A. *GreenFlag Construction: A Novel VT Construction*

With an eye on implementation, we introduce a novel construction algorithm for Varshamov-Tenengolts codes. We can construct VT(n) codes for any value of n; however, to provide fast and efficient encoding and decoding, we use powers of two, i.e., $n = 2^l$. Our encoding and decoding algorithms are based on the mathematics of VT codes, and we

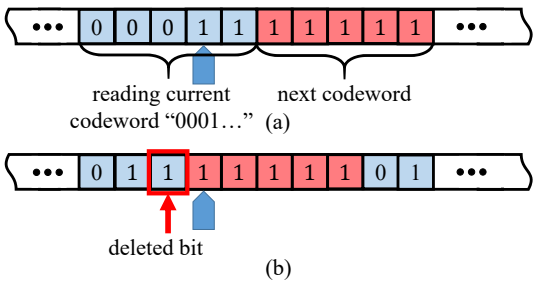

Fig. 6. The boundary problem

have proven them correct. Due to space constraints, proofs could not be included in this paper but are provided in a document available on-line [27].

1) Encoding

Suppose $n = 2^l$ and $k = n - l - 1$. We encode k data bits, $d_1, d_2, \ldots, d_k$, into a codeword c of length n such that it satisfies the checksum as described in Section IV. Such a code is referred to as GreenFlag$(n, k, delimiter_size)$.

We present the encoding algorithm and an example for $n = 8$ in Table II. In this example, $l = 3$ and we encode $k = n - l - 1 = 4$ data bits (1011) into a codeword of length $n = 8$. The encoding algorithm creates $l + 1$ check bits to ensure that the checksum holds. We index bit positions in a codeword from 1 to n (rather than 0 to $n - 1$).

The rate of this code is given by,

$$R(n) = \frac{n - log_2(n) - 1}{n}$$

which approaches 1 in the limit of large n.

2) Error-Free Decoding

Because of our novel construction, if the checksum is zero when reading a codeword, we can simply recover our error-free dataword by extracting all the bits that are not powers of two. Thus, this allows for efficient decoding in the common, error-free case.

However, as we already discussed, a silent shift error can result in a correct codeword but leave the wrong domain over the read/write port, thus corrupting subsequent accesses. Error correction requires that we both detect this situation and differentiate insertions from deletions.

B. *Greenflag's Delimiters: Categorizing Shift Errors*

We use specially crafted delimiters to identify silent shift errors and to distinguish insertions from deletions. Intuitively, the delimiters help us to separate codewords from each other in the presence of shift errors. We refer to a codeword that is concatenated with delimiter bits as an *extended codeword*.

TABLE II. ENCODING ALGORITHM WITH EXAMPLE

Encoding Algorithm	Example for $n = 8, l = 3$
Step 1: Start with a zero-vector c of length $n = 2^l$	$\boldsymbol{c} = 0\,0\,0\,0\,0\,0\,0\,0$
Step 2: Set positions that are not powers of two to data bits	$\boldsymbol{d} = 1011$ $\boldsymbol{c} = 0\,0\,\underline{1}\,0\,\underline{0}\,\underline{1}\,\underline{1}\,0$
Step 3: Calculate the checksum $\sum_{i=1}^{n} ic_i$	$\sum_{i=1}^{8} ic_i = 3 + 6 + 7 = 16$
Step 4: Set s to be the minimum value that needs to be added to the checksum to make it equal to 0 modulo $n + 1$	$s = 9 - mod(16,9) = 2$
Step 5: Set the $l + 1$ positions that are powers of two to the binary expansion of s. Start from c_1 and set it to be the MSB. Move all the way to c_n and set it to the LSB	$s = 2 = (0010)_2$ $c_1 = 0, c_2 = 1$ $c_3 = 0, c_4 = 0$ $\boldsymbol{c} = \underline{0}\,\underline{1}\,1\,\underline{0}\,0\,1\,1\,\underline{0}$

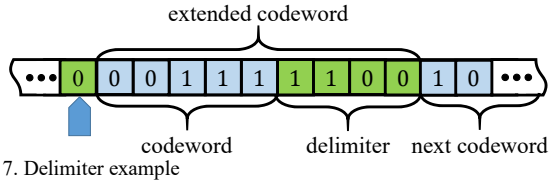

Fig. 7. Delimiter example

We choose delimiters to most efficiently achieve our goals, and tolerating greater numbers of shift errors requires longer delimiters. All delimiters must be functional when read from right to left or left to right because domains on a track can be shifted and read in both directions. In other words, delimiters should either be palindromic or have their less significant bits equal to the complement of the corresponding more significant bits.

We illustrate an example of extended codewords in Fig. 7. The example has 5-bit VT codewords and 4-bit delimiters of 1100. In the absence of shift errors, when we read an extended codeword, its last four bits will be the known bits 1100 of the delimiter.

Choosing the delimiter bits to use can be tricky, because of edge cases like a shift error in the delimiter bits themselves. Our crafted delimiters, with the help of the VT codes, can help us detect and correct shift errors even in such corner cases, and we exhaustively tested our scheme against every possible combination of shift errors in data bits and delimiter bits [27]. We now present two options, with different lengths, to show how the choice of delimiter impacts the ability to identify shift errors.

Delimiter-1 is the 4-bit pattern $d = (d_1, d_2, d_3, d_4) = 1100$. We use delimiter-1 along with the VT checksum to categorize shift errors as shown in Table III. Note that in Table III we do not include the observation of d_4, because any error that affects the last delimiter bit will be detected when we read the next extended codeword. Because delimiter-1 can detect some double shift errors (1 insertion plus 1 deletion) but not all (double deletion or double insertion), we classify a GreenFlag(*n,k,4*) code as a SECSED code for shift errors.

Delimiter-2 is a 6-bit delimiter with the pattern 111000. Table IV shows how to use the VT checksum and delimiter-2 to categorize shift errors. Unlike delimiter-1, delimiter-2 can detect and differentiate double deletions and double insertions. Although VT codes can only recover codewords from a single shift error, delimiter-2 (with the help of the VT checksum) enables GreenFlag to correct the relative position of the read/write port for up to double shift errors. Thus, double shift errors can now be detected and corrected by re-reading the extended codeword in the opposite direction. Note that VT codes enable us to correct single shift errors without the need to re-read an extended codeword and thus single shift error correction will be substantially faster than correcting

TABLE III. HOW DELIMITER 1100 IDENTIFIES ERRORS (X = 0 OR 1)

Observed delimiter bits				Decisions
d_1	d_2	d_3	Checksum	
1	1	0	0	no error
1	1	0	$\neq 0$	1 insertion & 1 deletion
1	0	0	$\neq 0$	1 deletion
X	1	1	$\neq 0$	1 insertion

TABLE IV. HOW DELIMITER 111000 IDENTIFIES SHIFT ERRORS (X = 0 OR 1)

Observed delimiter bits						Decision
d_1	d_2	d_3	d_4	d_5	Checksum	
1	1	1	0	0	0	**no error**
1	1	1	0	0	$\neq 0$	**1 deletion and 1 insertion**
1	1	0	0	0	X	**1 deletion**
X	1	1	1	0	X	**1 insertion**
1	0	0	0	X	X	**2 deletions**
X	X	1	1	1	X	**2 insertions**

TABLE V. DELETION CORRECTION WITH AN EXAMPLE

Deletion Correction Algorithm	Example for $n = 8, l = 3$
Step 1: Suppose a VT codeword $\boldsymbol{c} = (c_1, c_2, \dots, c_n)$ is stored in racetrack memory. Using GreenFlag we have detected a single deletion and thus we observe $\boldsymbol{c'} = (c'_1, c'_2, \dots, c'_{n-1})$.	$\boldsymbol{c} = 0\,1\,1\,0\,0\,1\,\underline{1}\,0 \rightarrow \boldsymbol{c'} = 0\,1\,1\,0\,0\,1\,0$
Step 2: Set ω to be the Hamming weight (number of 1's) of c'.	$\omega = 3$
Step 3: Calculate the checksum $\sum_{i=1}^{n} ic'_i$	$\sum_{i=1}^{8} ic'_i = 2 + 3 + 6 = 11$
Step 4: Set s to be the minimum amount that needs to be added to the checksum in order to make it 0 modulo $n+1$	$s = 7$
Step 5: If $s \leq \omega$, we restore the codeword by adding a 0 immediately to the left of the rightmost s 1's. Otherwise, we restore the codeword by adding a 1 immediately to the right of the leftmost $(s - \omega - 1)$ 0's.	$s > \omega \rightarrow$ restore the codeword by adding a 1 immediately to the right of the leftmost $(s - \omega - 1) = 3$ zeros to arrive at the correct codeword: 0 1 1 0 0 $\underline{1}$ 1 0.

TABLE VI. INSERTION CORRECTION WITH AN EXAMPLE

Insertion Correction Algorithm	Example for $n = 8, l = 3$
Step 1: Suppose a VT codeword $\boldsymbol{c} = (c_1, c_2, \dots, c_n)$ is stored in racetrack memory. Using GreenFlag we have detected a single insertion and thus we observe $\boldsymbol{c'} = (c'_1, c'_2, \dots, c'_{n+1})$.	$\boldsymbol{c} = 0\,1\,1\,\underline{0}\,0\,1\,1\,0 \rightarrow \boldsymbol{c'} = 0\,1\,1\,0\,0\,0\,1\,1\,0$
Step 2: Set ω to be the Hamming weight (number of 1's) of c'.	$\omega = 4$
Step 3: Calculate the checksum $\sum_{i=1}^{n} ic'_i$	$\sum_{i=1}^{8} ic'_i = 2 + 3 + 7 + 8 = 20$
Step 4: Set s to be the minimum amount that needs to be deducted from the checksum in order to make it 0 modulo $n+1$	$s = 2$
Step 5: If $s \leq \omega$, we restore the codeword by removing a 0 immediately to the right of the leftmost s 1's. Otherwise, we restore the codeword by removing a 1 immediately to the left of the rightmost $(s - \omega)$ 0's.	$s \leq \omega \rightarrow$ restore the codeword by removing a 0 immediately to the right of the leftmost $s = 2$ ones to arrive at the correct codeword: 0 1 1 ~~0~~ 0 0 1 1 0.

double shift errors. Overall a GreenFlag(*n,k,6*) code is a DECDED code for shift errors.

Note that GreenFlag combines delimiters and the VT checksum to categorize shift errors. Delimiters alone do not suffice to detect nor to determine the type of shift errors.

C. *Decoding in Presence of Shift Errors*

If a shift error is detected, we shift so that the correct domain is over the read/write port. If it is a double shift error, we need to re-read the entire extended codeword. In the case of a single shift error, we correct the erroneous bit values as explained next.

Correcting a single deletion error: This can be accomplished using the elegant algorithm proposed by Levenshtein [28, 29] and which we present in Table V alongside an example for *n*=8. The main idea is that we want to find a position at which inserting a bit will make the checksum zero. Using the decoding algorithm, we reconstruct the correct codeword. Then we recover the dataword as

discussed for the error free case. It is worth mentioning that while the decoding algorithm corrects the codeword, it does not reveal the exact position where the deletion occurred. In fact, in the example in Table V, if either of the 1's in positions 6 or 7 was deleted, the algorithm would have behaved in exactly the same way.

Correcting a single insertion error: Handling an insertion is very similar to handling a deletion and follows the same logic. We modify the Deletion Correction Algorithm to handle an insertion as given in Table VI. Again, it is worth mentioning that while the algorithm corrects the codeword, it does not reveal the exact position where the insertion occurred.

D. *Block Organization*

In order to minimize the access latency of a block, GreenFlag uses multiple tracks to store a single block similar to prior work [12, 30]. Assuming a 512-bit block, prior work uses 512 tracks to stripe a block at a bit granularity. GreenFlag stripes a block at an extended codeword granularity. For example, GreenFlag(8,4,6) uses 128 tracks that each stores a (8+6)-bit-long extended codeword. All 128 extended codewords can be read in parallel to retrieve a single block.

VI. Hardware Implementation

In this section, we discuss the hardware implementation of the encoding and decoding process of GreenFlag. Designing an encoder and decoder helps us analyze and understand the overheads of GreenFlag in terms of latency, energy and area.

We used Verilog to describe all of the hardware down to the RTL, and we then synthesized it using Synopsys design compiler with the 15nm CMOS technology node.

A. *Encoding*

1) Simple Implementation

To start the encoding process, we fill a FIFO queue that has a size of $n = 2^l$, i.e., the length of our codeword. We place the desired data bits in all the positions of the queue that are not powers of 2 as described in Section V. This process represents steps 1 and 2 of our encoding algorithm.

The content of the FIFO queue is then shifted to a full adder (FA) as seen in Fig. 8. The FA uses the shifted bits of the queue as its enable input. In other words, the FA performs an addition only when the shifted bit has a value of 1. The first input of the FA is driven by a counter that counts from 0 to $n-1$. Additionally the carry input of the FA is always set to 1 and thus we add the numbers from 1 to n. The second input is driven through a feedback loop from the output of the FA. The output is latched in order to allow for correct synchronization. Thus, we have now completed step 3 of our algorithm and calculated the desired *sum*.

We continue by calculating the $n+1$ modulo of the *sum*. To efficiently perform the modulo operation we exploit some observations. The calculated sum of step 3 is bounded by the follow expression.

$$\sum_{i=1}^{n} ic_i \leq \sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

This allows us to efficiently calculate the $n+1$ modulo operation with the following algorithm:

1. Divide sum with n and store the result in $quot\_tmp$.
2. Store the remainder in rem_tmp.
3. If $quot\_tmp \leq rem\_tmp$ then $remainder = rem\_tmp - quot\_tmp$.
4. Else $quot\_tmp = quot\_tmp - 1$; $rem\_tmp = rem\_tmp + n$; $remainder = rem\_tmp - quot\_tmp$

This algorithm performs division with n instead of $n+1$, which is a simple operation, because n is always a power of 2 (i.e., $n = 2^l$). Thus, $quot\_tmp$ can be calculated by just extracting the $l - \log_2 n$ most significant bits of the *sum*. Then rem_tmp is the $\log_2 n$ least significant bits. The modulo block implements this algorithm.

Now we can calculate s by just subtracting the calculated $remainder$ from $n+1$. Note that the result of this operation is already the desired binary expansion and thus we simply place the output bits in the codeword positions that are powers of 2. We have now completed steps 4 and 5 of the encoding algorithm.

A drawback of this implementation is that the delay overhead due to encoding depends on the length of the codeword. As the codewords become larger, in order to achieve better rate, the delay overhead becomes significant.

2) Performance Optimized Implementation

In order to mitigate the increase in delay overhead we slightly modify the encoding process by calculating *sum* in parallel. We split the FIFO queue into smaller queues of 8-bit length. Each queue is now connected to a separate FA that computes a *partial sum*. Multiple partial sums are computed in parallel. We then add all the *partial sums* together to get the final *sum*.

This optimization allows us to trade area for lower latency. We note that although the two implementations differ in power consumption, there are negligible differences in energy due to a significant reduction in latency. In other words, we now spend more power over less time.

B. *Decoding*

Decoding has some of the same steps as encoding. As we read our codeword from the racetrack, we calculate the *sum* as we did for encoding. We then perform the modulo operation to get the desired *Checksum.* Also, as we read the codeword we calculate its weight *ω.* We use the module "Delimiter" to store the delimiter bits. We then use the *Checksum* with the delimiter bits to decide if there was an error as well as the type of the error as discussed in Section V.

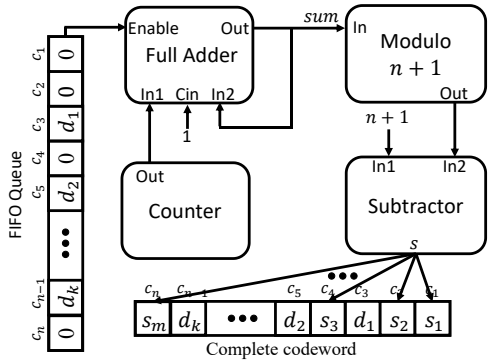

Fig. 8. Hardware implementation of the encoder

TABLE VII. HARDWARE IMPLEMENTATION OVERHEADS

	Encoding (optimized)					Decoding (error free)				
Codeword length (bits)	4	8	16	32	64	4	8	16	32	64
Cycles	1	12	14	15	16	11	15	23	39	71
Frequency (GHz)	1	1	1	1	1	0.8	0.8	0.8	0.8	0.8
Latency (ns)	1	12	14	15	16	8.75	8.75	8.75	8.75	8.75
Power (uW)	10	204	311	755	974	212	286	413	675	812
Energy (pJ)	0.01	2.45	4.35	11.33	15.58	0.24	5.36	11.87	32.91	72.07
Area (um^2)	11	326	646	1117	2704	652	1014	1361	1754	2321
Rate (w/ 6-bit delimiter)	0.1	0.286	0.5	0.684	0.814	0.1	0.286	0.5	0.684	0.814

The "Error Correction" module uses the *Checksum, ω,* and the error decision to decode and recover the correct codeword by performing steps 4 and 5 of the decoding process. From the correct codeword we can now extract the initial stored dataword by just reading the positions that are not powers of 2. The whole process is shown in Fig. 9.

C. *Analysis: Latency, Power, Energy, and Area*

We now present the results of evaluating the hardware that we implemented in Verilog and synthesized with Synopsys Design Compiler. Table VII presents latency, power, energy, and area results for various codeword lengths (n) for the optimized encoding process and the error-free decoding process. These results do not include the actual costs to read or write a codeword; in Section VII we use these results to evaluate GreenFlag as a whole.

As expected, overheads tend to increase as n increases. However, we observe that the latency for error-free decoding is constant with respect to n. That happens because part of the decoding process is done in parallel with reading the codeword from the racetrack.

Even though error-free performance is more critical, we also study decoding performance in the presence of errors. For single shift errors, the latency depends on the position of the error. On average, the expected decoding latency is 8.75ns (the error-free latency) plus $n/2$ cycles (at 0.8GHz) to shift half-way through the codeword. For $n = 32$, that latency equals 28.75ns. For double shift errors, the latency is twice the error-free latency plus the product of n and the latency to re-read each bit (i.e., 3.7ns to shift and 2.1ns to read, as shown in Table VII). For $n = 32$, this latency equals 203ns.

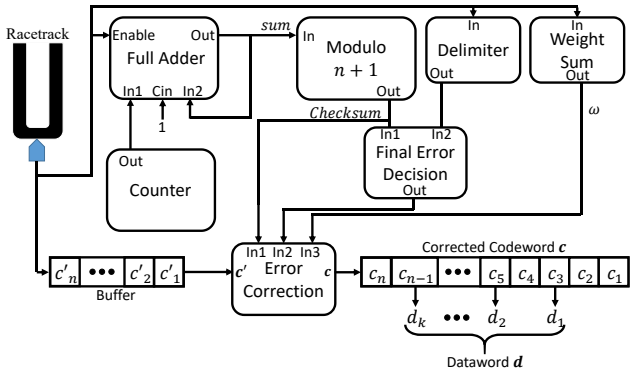

Fig. 9. Hardware implementation of the decoder

Lastly, we note that a designer could choose to use GreenFlag only for shift error detection (and not correction) by (a) simply discarding the "Error Correction" module in Fig. 9 and (b) always using re-read attempts to recover from shift errors. Although that solution would save some area (~45%) and energy (~12%) for the decoding hardware, it would cause all shift errors (single and double) to incur the latency of double shift errors. Because single shift errors are far more common [12], we do not consider this option during our evaluation.

VII. EVALUATION

In this section we evaluate GreenFlag. Recall that our goal is to determine the viability of Racetrack with effective error tolerance, not to promote Racetrack as necessarily the best option, and our analysis is thus more of a limit study than a cycle-accurate comparison against other schemes.

We assume a best-case racetrack that has the minimum possible read, write, and shift latencies, based on results from prior work on quantitative analysis for racetrack memory [31]. We chose the minimum latencies so that we maximize the significance of GreenFlag's overheads. We summarize the characteristics of the best-case racetrack in Table VIII. We combine racetrack's latency and energy overheads with the results of Section VI to calculate the overall overheads of GreenFlag. For all the results, we use delimiter-2, the 6-bit delimiter that provides DECDED shift error protection.

A. *Block Read and Write Latencies*

For 64-byte blocks, Fig. 10 presents the block read and write latency of GreenFlag for different rates (i.e., for different codeword lengths). These latencies include shift, read/write and decoding/encoding delays. For example, GreenFlag(8,4,6) has 8-bit codewords and 6-bit delimiters, and reading an extended codeword requires 8+6 shifts and 8+6 reads; its rate is 0.286.

The results of Fig. 10 show that racetrack memory with GreenFlag cannot achieve the latency expected of an L1 cache and probably not even the latency of a last-level cache (LLC).

TABLE VIII. BEST-CASE RACETRACK

Operation (per bit)	Latency (ns)	Energy (pJ)
Read	3.7	224
Write	10.2	998
Shift	2.1	124

GreenFlag(4,1,6) can provide the lowest read latency (67ns) but with a huge overhead of 0.1 rate. Even if we use GreenFlag(4,1,4) (that provides SECSED shift protection) read latency will only drop to 56ns and the rate will be 0.125.

However, the results show that racetrack with GreenFlag is a viable option for main memory and storage devices. GreenFlag(32,26,6) and GreenFlag(16,11,6), with rates of 0.684 and 0.5, respectively, can provide latencies comparable to modern DRAM designs like DDR3 (~100ns), while GreenFlag(64,57,6) can be used as an extremely low latency storage device with a rate of 0.814. While these rates may still seem somewhat low we remind the reader that GreenFlag is, to the best of our knowledge, the only shift error solution that can be implemented with only one read/write port. Compatibility with a single port makes GreenFlag the only coding scheme that can be used with 3D racetracks and benefit from their density compared to 2D racetracks [5, 11, 15, 16].

B. *Bandwidth*

In Fig. 11 we present GreenFlag's maximum data bandwidth per track. Bandwidth per track is a more insightful metric than aggregate bandwidth, at this early stage in the development of racetrack memory, because it eliminates orthogonal issues like I/O bottlenecks and open questions about how large-scale racetrack will be organized.

We calculate the bandwidth per track as the average number of data bits we can read or write per track per second, including delimiter and encoding/decoding overheads. For comparison, we also include the bandwidth of prior work on tolerating shift errors (HiFi) [12]. Note that HiFi's bandwidth is constant (i.e., not a function of n) for reasons explained in Section VIII.

We observe that, as the rate increases, the read and write bandwidth per track of GreenFlag increases. This result occurs because we are accessing more data bits per codeword, while the decoding and encoding overheads remain almost constant. GreenFlag achieves up to 1.86× more read bandwidth per track compared to HiFi, while providing similar write bandwidth. The reason is that HiFi must perform a write after every shift operation, which limits how many bits it can stream per second.

C. *Energy*

We calculate the average energy per bit for read and write operations. For GreenFlag this is calculated by amortizing the cost of an extended codeword, including the decoding and encoding energy overheads, per bit. Fig. 12 shows that the energy overheads of GreenFlag decrease as rate increases. Additionally, we observe that the energy overheads are significantly lower than prior work [12] (up to 6.6×).

D. *Mean Time to Failure (MTTF)*

GreenFlag's ability to tolerate an error when reading a given extended codeword depends on the probability of a shift error on any given shift and the specific GreenFlag code used. GreenFlag's overall ability to tolerate errors and avoid silent data corruption (SDC) also depends on the memory bandwidth demanded. The dependence on bandwidth distinguishes racetrack memory from other technologies; as

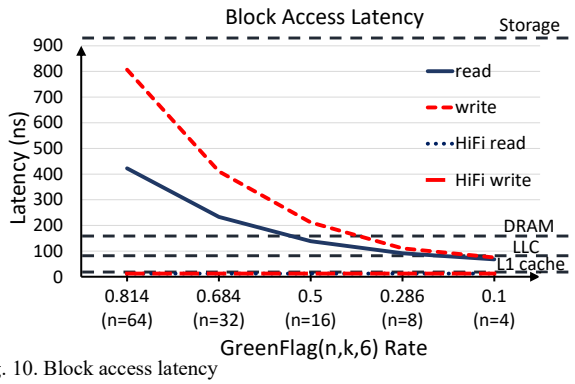

Fig. 10. Block access latency

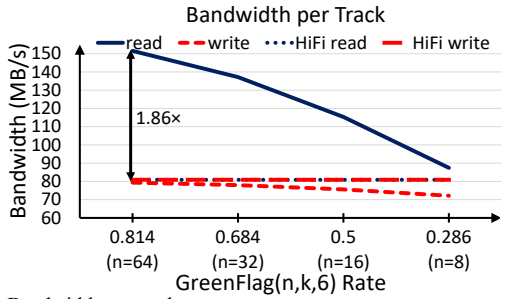

Fig. 11. Bandwidth per track

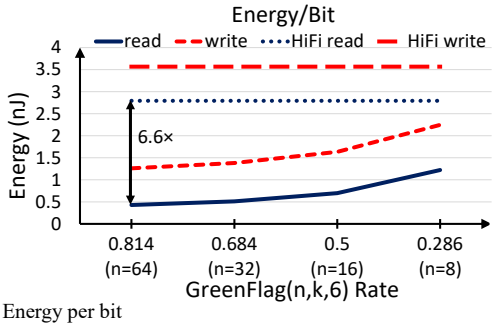

Fig. 12. Energy per bit

also noted by prior work [12], racetrack errors only occur during shifts and thus the more the racetrack is used (shifted), the more possible opportunities for errors.

The expected probability of a single shift incurring an error is not well known. Prior work [12] has estimated shift error probabilities based on models and simulations, but currently there are no empirical results. Thus, we test GreenFlag over a wide range of single shift error probabilities ranging from 10^{-6} to 10^{-9}. We account for both double shift errors and the possibility of multiple single shift errors in the same codeword.

We define failure as the occurrence of a silent data corruption (SDC). That is, three or more shift errors occur while reading a single codeword. We then analytically calculate the probability of failure per extended codeword (P_f). This probability depends on the length of the codeword and the single shift error probability, but not the bandwidth demand (i.e., we assume there is demand for exactly one extended codeword). Fig. 13 presents the failure probability per extended codeword for different GreenFlag codes and different single shift error rates; note that values on y-axis are in decreasing order. We observe that the failure probability increases linearly as the codeword length increases.

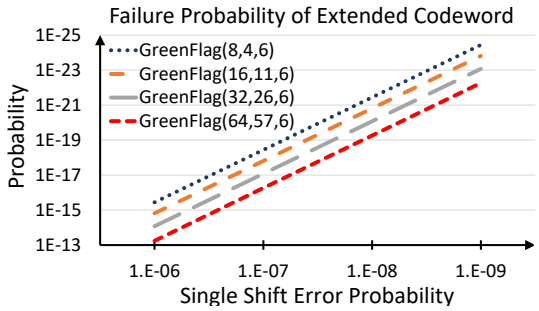

Fig. 13. SDC error rate per extended codeword

To calculate overall MTTF (across all tracks) we need to know how often a shift operation occurs. Thus, we assume different shift intensities that represent peak bandwidths for a variety of memory technologies like DDR3, SSDs, and 3D-DRAM. We can now calculate the MTTF as,

$$MTTF = \frac{1}{P_f} \times \frac{n}{Bandwidth} \times \frac{1}{3.154 \times 10^7}$$

where 3.154×10^7 is the number of seconds in one year and n is the length of a codeword.

Fig. 14 presents the MTTF for GreenFlag(64,57,6) and GreenFlag(32,26,6) for different bandwidths. We set a desired MTTF goal of 10 years. For very high shift error probabilities (i.e., greater than 10^{-7}), it is difficult to achieve that goal. However, for lower probabilities we can achieve MTTFs well beyond 10 years. Overall Fig. 14 can be used to extrapolate MTTFs for any single shift error probability and can be used as a guideline when designing reliable memory systems based on racetrack memory.

VIII. Prior Work

Prior work on handling shift errors in racetrack memory [12, 13, 32] uses additional read/write ports to access multiple bits at the same time. These extra ports preclude using the denser 3D racetrack layout, but they enable detecting and correcting multiple shift errors. HiFi [12] and its extended work [32] address bit deletions in racetrack memory by adding pattern bits at both edges of each track as well as additional read and write ports. The main idea is to read the pattern in parallel with the data bits. If the pattern does not match what was expected, bits are shifted in the reverse direction and the memory is re-read to recover the bits correctly. As bits are shifted during consecutive reads, some of the pattern bits are "shifted out" of the track and virtually get deleted. Thus, for every shift in racetrack, one data bit and two pattern bits are read, while at the same time a new pattern bit is written. Chee at al. [13] propose placing multiple ports with a specific number of domains between any two consecutive ports. They encode the data in a way that a shift error can be detected from observing differences when reading the same data from two nearby ports. This work however ignores the fact that domains are virtually deleted if shifted out of the track, something that can happen as domains are shifted from the first to the second port.

There is also prior work on the use of Varshamov-Tenengolts codes [33] to correct segmented errors on deletion and insertion channels in which deletions *or* insertions (but not a mixture of two) could occur. This is a more restrictive

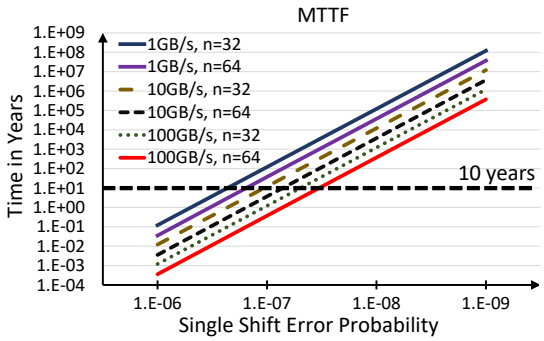

Fig. 14. MTTF for different implementations/bandwidths

model than racetrack memory in which deletions and insertions could both occur. Moreover, if we input n bits to deletion and insertion channels with segmented errors, the output would be a sequence of length n' from which we can easily deduce the number of shift errors. Again, this differs from racetrack memory where the memory controller always provides the desired number of bits, n, regardless of whether an error occurs. Moreover, in this work, exhaustive search is used to find codewords of a given length which makes implementation infeasible.

Additionally, there is prior work on utilizing racetrack memory to design caches and memory systems [3, 30, 34], but none of this work considers the system's fault tolerance.

IX. Conclusion

We have presented GreenFlag, the first error tolerance scheme for 3D racetrack memory. GreenFlag combines a novel construction for Varshamov-Tenegolts codes with specially crafted delimiter bits to detect, categorize and correct shift errors. Additionally, we designed and synthesized hardware for encoding and decoding so that we could estimate latency and energy overheads and, more significantly, analyze the viability of racetrack for different levels of the memory hierarchy. Based on current technology, it appears that fault tolerant 3D racetrack memory is not attractive for caches but a promising option for main memory and storage.

X. Acknowledgments

This material is based on work supported by the National Science Foundation under grants CCF-142-1177 and CCF-171-7602.

References

[1] S. Mittal, "A survey of techniques for architecting processor components using domain wall memory," ACM Journal on Emerging Technologies in Computing Systems, 2016, no. 2, p.29.

[2] L. Thomas, S.-H. Yang, K.-S. Ryu, B. Hughes, C. Rettner, D.-S. Wang, C.-H. Tsai, K.-H. Shen, and S. S. Parkin, "Racetrack memory: a high-performance, low-cost, non-volatile memory based on magnetic domain walls," in IEEE International Electron Devices Meeting (IEDM), pp. 24–2, 2011.

[3] Z. Sun, W. Wu, and H. Li, "Cross-layer racetrack memory design for ultra high density and low power consumption," in 50th ACM/EDAC/IEEE Design Automation Conference (DAC), pp. 1–6, 2013.

[4] S. Parkin and S.-H. Yang, “Memory on the racetrack,” Nature nanotechnology, vol. 10, no. 3, pp. 195–198, 2015.

[5] S. S. Parkin, M. Hayashi, and L. Thomas, “Magnetic domain-wall racetrack memory,” Science, vol. 320, no. 5873, pp. 190–194, 2008.

[6] M. K. Qureshi, “Pay-as-you-go: low-overhead hard-error correction for phase change memories,” in the 44th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 318–328, 2011.

[7] Y. Zhang, W. Zhao, D. Ravelosona, J.-O. Klein, J. Kim, and C. Chappert, “Perpendicular-magnetic-anisotropy cofeb racetrack memory,” Journal of Applied Physics, vol. 111, no. 9, p. 093925, 2012.

[8] J. J. Yang, D. B. Strukov, and D. R. Stewart, “Memristive devices for computing,” Nature nanotechnology, vol. 8, no. 1, pp. 13–24, 2013.

[9] M. Hayashi, L. Thomas, R. Moriya, C. Rettner, and S. S. Parkin, “Current-controlled magnetic domain-wall nanowire shift register,” Science, vol. 320, no. 5873, pp. 209–211, 2008.

[10] W. Zhao, Y. Zhang, H. Trinh, J. Klein, C. Chappert, R. Mantovan, A. Lamperti, R. Cowburn, T. Trypiniotis, M. Klaui, et al., “Magnetic domain-wall racetrack memory for high density and fast data storage,” in Solid-State and Integrated Circuit Technology (ICSICT), 2012 IEEE 11th International Conference on, pp. 1–4, 2012.

[11] Y.P. Ivanov, A. Chuvilin, S. Lopatin and J. Kosel, "Modulated magnetic nanowires for controlling domain wall motion: toward 3D magnetic memories," ACS NANO, 10(5), pp.5326-5332, 2016.

[12] C. Zhang, G. Sun, X. Zhang, W. Zhang, W. Zhao, T. Wang, Y. Liang, Y. Liu, Y. Wang, and J. Shu, “Hi-Fi playback: Tolerating position errors in shift operations of racetrack memory,” in Proceedings of ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA), pp. 694–706, 2015.

[13] Y. M. Chee, H. M. Kiah, A. Vardy, V. K. Vu, and E. Yaakobi, “Coding for racetrack memories,” arXiv preprint arXiv:1701.06874, 2017.

[14] R. R. Varshamov and G. M. Tenengolts, “Codes which correct single asymmetric errors,” Avtomatika i Telemekhanika (in Russian), vol. 26, no. 2, pp. 288–292, 1965.

[15] M. Hirofumi, H. Fukuzawa, A. Kikitsu and Y. Fukuzumi, "Magnetic Memory Device and Method of Magnetic Domain Wall Motion," U.S. Patent 8,792,271, issued July 29, 2014.

[16] D. Sanz-Hernández, R.F. Hamans, J.W. Liao, A. Welbourne, R. Lavrijsen and A. Fernández-Pacheco, "Fabrication, detection and operation of a three-dimensional nanomagnetic conduit," arXiv preprint arXiv:1706.03710.

[17] N. J. Sloane, “On single-deletion-correcting codes,” Codes and Designs, pp. 273–291, 2002.

[18] R. G. Gallager, “Sequential decoding for binary channels with noise and synchronization errors,” tech. rep., DTIC Document, 1961.

[19] R. L. Dobrushin, “Shannon’s theorems for channels with synchronization errors,” Problemy Peredachi Informatsii, vol. 3, no. 4, pp. 18–36, 1967.

[20] S. N. Diggavi and M. Grossglauser, “On transmission over deletion channels,” in the Annual Allerton Conference on Communication Control and Computing, vol. 39, pp. 573–582, 2001.

[21] A. Kavcic and R. Motwani, “Insertion/deletion channels: Reduced-state lower bounds on channel capacities,” in IEEE International Symposium on Information Theory, pp. 229–229, 2004.

[22] S. Diggavi, M. Mitzenmacher, and H. Pfister, “Capacity upper bounds for deletion channels,” in Proceedings of the International Symposium on Information Theory (ISIT), pp. 1716–1720, 2007.

[23] M. Mitzenmacher, “A survey of results for deletion channels and related synchronization channels,” Probability Surveys, vol. 6, pp. 1–33, 2009.

[24] J. Ullman, “On the capabilities of codes to correct synchronization errors,” IEEE Transactions on Information Theory, vol. 13, no. 1, pp. 95–105, 1967.

[25] G. Tenengolts, “Nonbinary codes, correcting single deletion or insertion (corresp.),” IEEE Transactions on Information Theory, vol. 30, no. 5, pp. 766–769, 1984.

[26] F. Palunˇci´c, K. A. Abdel-Ghaffar, and H. C. Ferreira, “Insertion/deletion detecting codes and the boundary problem,” IEEE Transactions on Information Theory, vol. 59, no. 9, pp. 5935–5943, 2013.

[27] G. Mappouras, A. Vahid, R. Calderbank and D. J. Sorin "Support Material for GreenFlag: Protecting 3D-Racetrack Memory from Shift Errors," https://www.dropbox.com/s/o3pf0emvsa8onpp/Racetrack_DSN_19_Support_Material.pdf?dl=0, accessed: 2019-3-8.

[28] V. I. Levenshtein, “Binary codes capable of correcting deletions, insertions and reversals,” in Soviet physics doklady, vol. 10, p. 707, 1966.

[29] V. Levenshtein, “Binary codes capable of correcting spurious insertions and deletions of ones,” Problems of Information Transmission, vol. 1, no. 1, pp. 8–17, 1965.

[30] R. Venkatesan, S.G. Ramasubramanian, S. Venkataramani, K. Roy and A. Raghunathan, "Stag: Spintronic-Tape Architecture for GPGPU Cache Hierarchies," in International Symposium on Computer Architecture (ISCA), ACM/IEEE, pp. 253-264, 2014.

[31] C. Zhang, G. Sun, W. Zhang, F. Mi, H. Li, and W. Zhao, "Quantitative modeling of racetrack memory, a tradeoff among area, performance, and power," in 20th Asia and South Pacific Design Automation Conference (ASP-DAC), pp. 100-105, 2015.

[32] X. Wang, C. Zhang, X. Zhang and G. Sun, "np-ECC: Nonadjacent Position Error Correction Code For Racetrack Memory," in International Symposium on Nanoscale Architectures (NANOARCH), pp. 23-24, IEEE, 2016.

[33] Z. Liu and M. Mitzenmacher, “Codes for deletion and insertion channels with segmented errors,” IEEE Transactions on Information Theory, vol. 56, no. 1, pp. 224–232, 2010.

[34] R. Venkatesan, V. Kozhikkottu, C. Augustine, A. Raychowdhury, K. Roy, and A. Raghunathan, “TapeCache: a high density, energy efficient cache based on domain wall memory,” in ACM/IEEE international symposium on Low power electronics and design, pp. 185–190, 2012.