1 Introduction

The continued scaling of photolithographic fabrication techniques down to 32 nanometers and beyond faces enormous technology and economic barriers. Self-assembled devices such as silicon nanowires or carbon nanotubes show promise to not only achieve aggressive dimensions, but to help address power and other gating issues in system architecture, while potentially helping contain rampant increases in fabrication capital costs. However, assembling high-quality, large-scale nanoelectronic circuits (e.g., with Langmuir-Blodgett or related methods) has proven challenging. Among the major challenges are extremely high defect and fault rates in assembled devices. Apart from fabrication errors, nanoscale devices are also more prone to soft errors than microscale devices. Current-day microscale devices (e.g., gates, PLAs, memories) constructed using top-down lithographic techniques have error rates of less than 1% [10]. But computing and storage components built using nanoscale elements (e.g., bistable and switchable organic molecules, carbon nanotubes, single-crystal semiconductor nanowires) may have an order of magnitude higher rates of faults (as high as 10%) [5, 8].

We consider static defects and soft errors separately. Static defects can be handled using testing and reconfiguration [21], though this presents increasing challenges as technology scales. Soft error correction is critical for different nanoscale devices, performing storage (e.g., nanomemory), computation (e.g., nano-ALU) or communication (e.g., nanoscale signal transmitters and receivers).

In this paper, we focus on error correction for nanoscale memory. We will concentrate on an architecture that uses nano-PLA blocks and simple nanogates (e.g., majority gate, nand/nor gates) as design components [7]. Because of high soft-error rates, our envisioned nanomemories would also need to employ online error correcting codes (ECCs), as most modern memory subsystems already do [10, 13]. However, for nanomemories with high fault rates, a new type of error correction is desired, since conventional ECC techniques are not directly applicable.

The encoders and decoders for Hamming and Hsiao codes, for example, have low encoding and decoding complexity, but also have relatively low error-
correcting capacity (e.g., Hamming is single error-correcting, double error-detecting). To achieve higher error-correcting capability, codes like Reed-Solomon or BCH require more sophisticated decoding algorithms, which would need either (a) complex algebraic decoders that can decode in fixed time – the designs for these complex operations (e.g., floating point operations, logarithms) would be difficult to implement using nanoscale PLAs, which favor simple regular designs, or (b) simpler graphical decoders, that use iterative algorithms (e.g., belief propagation) – these typically need more computation time, and hence would not be fast enough for at-speed ECC operations for nanomemory [16, 4].

For these reasons, we desire an error-correcting system that has (1) high error detecting and correcting ability, to tolerate relatively high soft error rates; and (2) sparse encoding, decoding and checker circuits, so that they can be synthesized using simple nanoscale hardware. Additional properties that are desirable for some applications include (a) modular encoder and decoder blocks, which can simplify and shrink hardware design; (b) systematic code structure, which will cleanly partition the information and code bits in the memory; and (c) dynamic error-correcting capability, to enable engineering the trade-off between error correction and system performance. In this report, we focus on the property of dynamic error correction.

As the lifetime of a part increases, the faults decrease during the normal operation of the circuit as compared to the infant mortality rate, and later increase again as the part ages. One would typically want the ECC circuit for the nanomemory to have higher error-correcting capacity during the initial and final stages, but disable unused parts of the ECC encoder/decoder circuit while operating at a lower error-correcting rate during normal operation, thereby saving power.

In this paper, we use low-density parity check (LDPC) codes [12] for ECC in nanomemories. We propose the use of a variant of a particular type of LDPC code, Euclidean Geometry (EG) LDPC [16], which is built using special structures of finite Euclidean Geometry. Various types of EG-LDPC codes have the different properties listed before, e.g., type-I codes are systematic, type-II codes have encoding and parity check matrices with regular modular structure, Gallager codes have properties that enable their error correction rate to be changed dynamically. More details about these codes and their properties are given in Sections 3 and 4. The sparseness (enabling low fan-in circuit implementation) and low computational overhead of decoding EG-LDPC codes make them easy to implement using nanoscale hardware.

Various nanodevices have been proposed in the literature (e.g., nanowire-based PLA, quantum nanodots) to serve as the basic building block of memory and computation elements. In this paper, we propose the design of a nanomemory array using nanowire-based-PLA components [9]. We analyze the properties of EG-LDPC codes that make it suitable as ECC for nanomemory, and suggest how the coder and decoder circuits can be efficiently fabricated using nano-PLA memory units and gates. We also consider that the encoder and checker circuits themselves can suffer faults, since they may be fabricated out of fault-prone nanoscale components. We provide an analysis of fault detection and correction
capacities of nanomemories with EG-LDPC codes, and design an overall system architecture based on nano-PLA building blocks.

2 Background

We outline concepts that will be useful in understanding our main design and architecture for ECC in nanomemory based on EG-LDPC codes.

2.1 LDPC codes

LDPC codes have several advantages, which have made them popular in many communication applications: (1) low density of the encoding matrix, (2) easy iterative decoding, (3) generating large code words that can approach Shannon’s limit of coding [16].

An LDPC code is defined as the null space of a parity check matrix \( H \) that has the following properties [16]:

1. Each row has \( \rho \) number of 1’s.
2. Each column has \( \gamma \) number of 1’s.
3. The number of 1’s that are common between any two columns (\( \lambda \)) is no greater than 1, i.e., \( \lambda = 0 \) or 1.
4. Both \( \rho \) and \( \gamma \) are small compared to the length of the code and the number of rows in \( H \).

As both \( \rho \) and \( \gamma \) are very small compared to the code length and the number of rows in the matrix \( H \), \( H \) has a low density of 1’s. Hence \( H \) is said to be a low-density parity check matrix and the code defined by \( H \) is said to be a low-density parity check code.

The density of \( H \) (\( r \)) is defined to be the ratio of the total number of 1’s in \( H \) to the total number of entries in \( H \) — in this case \( r = \rho/n = \gamma/J \), where \( J \) is the number of rows in \( H \). This kind of LDPC code is said to be a \((\gamma, \rho)\)-regular LDPC code. If the weights of all the columns or rows in \( H \) are not the same, then it is called an irregular LDPC code.

2.2 EG-LDPC codes

In recent work [26], Euclidean Geometry (EG) constructions over \( GF(2^s) \) have been used to construct \( H \)-matrices for LDPC codes. Using \( EG(m, 2^s) \), where each point in the geometrical space can be represented by an \( m \)-tuple over \( GF(2^s) \), the \( H \)-matrix can be interpreted as an incidence matrix — every column of the matrix represents a point in this space, every row represents a line, and every entry of 1 in the matrix represents that the corresponding row line is incident on the column point. Figure 1 shows an example of the EG-LDPC
code. This connection to finite Euclidean Geometry has advantages for our application:

(1) Using the structure of $EG(m,2^s)$, the LDPC $H$-matrices can be constructed quite efficiently, since this matrix has some useful properties. For example, for the Gallager code (see Section 3 for details), every $H_i$ component in the matrix represents a set of parallel lines (a bundle), which can be easily enumerated using Galois Field (GF) operations. This obviates the necessity of a search algorithm for designing the $H$-matrix. Instead, for any size of the memory, we can design the corresponding $H$-matrix efficiently using GF operations. Similar advantages exist in constructing other EG-LDPC codes, e.g., type I, type II.

(2) The regular geometric structure makes the corresponding LDPC code decodable using a multistep majority decoder, thus making it unnecessary to have iterative decoders [16]. As outlined in Section 1, this is one of the properties we desire in ECC for nanomemory, to make the decoding fast and thereby have low latency overhead for normal memory load/store operations.

Figure 1: Euclidean Geometry and the corresponding EG-LDPC $H$ matrix formed from the incidence matrix of the lines on the points in the geometry.
2.3 Decoding

Since EG-LDPC is a finite geometry cyclic code [16], it can be decoded effectively using one-step majority logic (ML) decoding. Here, we explain how that decoding is performed.

Let the $n$-bit faulty code word $r$ be the sum of the original code word $c$ and the error word $e$, where $e$ is 1 at the locations at which the errors/faults have occurred, 0 otherwise:

$$r = c + e.$$

To find whether a particular error bit $e_i$ is 1 or 0, the decoder first calculates parity equations, each of which calculates a check sum on some bits of $r$. A set of $J$ parity equations is orthogonal on $e_i$ if each of the $J$ parity equations check $e_i$ (i.e., $e_i$ is included in the check sum of each parity equation), but no other error bit is checked by more than one parity equation. If a set of $J$ parity equations is orthogonal on $e_i$, the ML decoding rule can be applied to decode $e_i$: $e_i$ is decoded to 1 if the majority of the parity check sums of the $J$ check sums is 1, otherwise $e_i$ is 0.

In EG-LDPC, for every bit position $i$ there is a set of $\gamma$ rows $A_i$ in $H$ that is orthogonal on $i$, where

$$A_i = \{h_{i1}^1, \ldots, h_{i\gamma}^1\},$$

i.e., the $i^{th}$ component of each row in $A_i$ is 1 and two rows in $A_i$ have a common 1 in no other positions. This signifies that the error bit in the $i^{th}$ position is ML decodable. For the $i^{th}$ bit position, a set of $\gamma$ syndrome equations $S_i$ is formed using the rows in $A_i$, where

$$S_i = \{s_j^i = e.h_j^i : h_j^i \in A_i, \text{for } 1 \leq j \leq \gamma\}$$

Correct decoding of $e_i$ using the ML decoding rule, applied on the outputs of the syndrome equations $S_i$, is guaranteed if the number of errors in $e$ is less than $\gamma/2$. This can be repeated for every bit position $i$ to estimate the error word $e$.

In this paper, we consider one-step ML decoding. However, the performance of the hard-decision one-step ML decoding can be further improved using Gallager’s iterative bit-flipping (BF) algorithm [12], which is able to correct errors when the number of errors exceeds $\gamma/2$. BF uses simple comparison, sum, and majority operations and is therefore easier to implement using nano-PLAs than the popular sum-product algorithms for decoding LDPC codes using belief-propagation, which requires real-number arithmetic and computation of logarithms.

2.4 Nano-PLA Architecture

In this work, we refer to the nanoarchitecture based on Programmable Logic Arrays (PLAs), as proposed by Dehon et al. [7]. He proposes to build a two-plane PLA with silicon nanowires. Figure 2 shows the proposed architecture. Nanowires can be aligned closely in a single orientation with the help of flow
techniques and then they can be rotated and repeated so that multiple layers of nanowires form crossbar arrays with switchable diodes at the crosspoints. These arrays can serve as memory cores, programmable wired-OR planes, and programmable crossbar interconnect arrays. As Figure 2 shows, the architecture is a combination of two coupled NOR-NOR planes, which can be also regarded as AND-OR PLA using DeMorgan’s Laws and complementation. In Figure 2 each of the two interconnected logic planes consists of a programmable wired-OR array followed by two restoration arrays; the first inverts the OR logic and the second serves as a noninverting buffer. Stochastic decoders are formed using vertical microscale wires A0-A3, which address the horizontal nanowires. Overall, this simple PLA structure serves as a universal logic structure as it supports AND, OR, and NOT operations.

![Figure 2: Nano-PLA architecture [7].](image)

The defects that occur in this kind of architecture are mainly of two types – (1) wire, where the wire is either functional or defective and (2) non-programmable crosspoint, where the crosspoint cannot be programmed into the on state, or the crosspoint may be shorted into an on state. For static defects, Naemi et al. [21] proposed a greedy algorithm for mapping a logic function on a nano-PLA with defective nanowires. In this work, we consider how dynamic errors can be handled using ECC.

## 3 Dynamic LDPC in Nanomemory

Dynamic ECC is a property of EG-LDPC codes that we will explore here in detail. We begin by outlining the motivation for having dynamic codes in nanoarchitectures.
3.1 Motivation

Soft error rates vary over the lifetime of complex electronic components. The “bathtub curve” of failure rates classically begins with a very high rate of infant mortality, then smooths to a relatively low rate of failure for some period of time, and then slowly rises toward the end of the life cycle. In addition, environmental stresses may change the expected fault rate (e.g., temperature, radiation levels, nearby radio transmission). Classically, one must design error correcting codes (ECCs) to tolerate the maximum failure rate expected over the entire lifetime of a device, resulting in “wasteful” tolerance of “too many” faults during the bulk of the component’s lifetime.

At the highest level, the work presented here aims to enable controlled reduction in fault tolerance during a part’s lifetime. More precisely, we work toward enabling the engineering of defect and fault tolerance, traded against system power and other key parameters. For example, if we are able to diagnose and reconfigure subcomponents after a time of high-fault-rate infant mortality, and we expect fault arrival rates to be greatly reduced for a significant period of time, we would like to turn off some of the power-hungry ECC circuits. Later, if fault rates were to climb again, we would wish to turn those ECC circuits back on.

In some cases, fault arrival rates are predictable. Some device families show a bathtub curve of fault arrival rates fairly smoothly [10]. In some systems, fault rates follow a mission profile (e.g., altitude), and in others fault rates are strongly affected by easily observable environmental conditions (e.g., temperature). Also, if a system is able to detect the frequency and severity (number of bits) of errors in encoded words, and fault rates change slowly (with respect to clock cycle), then the system may be able to predict future fault rates to some degree. In these cases, we wish to enable the dynamic control of fault tolerance, which may allow one to shut down a subset of ECC circuits to save power.

3.2 Proposed Approach

For dynamic ECC, we will use a particular type of LDPC code – Gallager code – which is the original LDPC code proposed by Gallager [12]. The advantage of this code is that the decoding parity check matrix (H-matrix) of this code can be expressed in a special form:

\[ H^T = (H_1^T H_2^T \ldots H_\gamma^T), \]

where \( H_1 \) is a matrix in the standard systematic form \([I : A]\) [4], and \( H_2, \ldots, H_\gamma \) are permutations of \( H_1 \). A \((\gamma, \rho)\)-regular Gallager code has a parity check matrix with column weight \( \gamma \) and row weight \( \rho \), and can correct as many as \( \gamma/2 \) errors. The number of rows in the \( H \)-matrix is \( J = k \times \gamma \), and the number of columns is \( n = k \times \rho \).

In dynamic ECC, one would typically want the ECC circuit for the nanomemory to have higher error correcting capacity during the initial and final stages, since faults decrease during the normal operation of the circuit as compared
Table 1: Different error correcting capability of variants of 64-bit code built using $EG(2, 2^4)$.

<table>
<thead>
<tr>
<th>Code</th>
<th>$\gamma$</th>
<th>ECC capacity</th>
</tr>
</thead>
<tbody>
<tr>
<td>(64,49)</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>(64,45)</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>(64,41)</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>(64,40)</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>(64,39)</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>(64,38)</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>(64,37)</td>
<td>8</td>
<td>4</td>
</tr>
</tbody>
</table>

to the high mortality rate during device infancy and later aging. Since the $H$ matrix of the Gallager code is modular, the controller can selectively enable modules of the ECC circuit at different times of the operation, so that $\gamma$ is higher during the initial and final phases and low during the middle phase of operation. The corresponding encoder can be modified accordingly, if reconfigurable nano-PLA hardware is used. This capability of changing the error correcting capacity of the encoder and decoder gives the required dynamic error-correcting capability.

For example, for an EG-Gallager code of length $n = 64$ constructed based on the Euclidean Geometry $EG(2, 2^4)$, the different error correction capacities for varying values of $k$ and $\gamma$ are shown in Table 1. As the ECC requirement decreases from 4 to 1, $k$ changes from 37 to 49 — this means that we can pack more bits into the same memory code word, since $n$ remains the same. Such a code can be used very effectively in our dynamic coding scheme.

The selective enabling and disabling of decoder blocks by the controller can give some power savings during normal operations. However, significant power savings are possible only if parts of the memory can be switched off, too. For that, we propose a memory repacking scheme in our memory bank architecture, details of which are described next.

### 3.3 Architecture

Figure 3 shows the overall system architecture of the nanomemory with EG-LDPC ECC. During a write operation, the incoming word to be stored in memory is encoded by the encoder and the code word is stored in memory. During a read operation, a code word is retrieved from the memory, checked by the checker unit, and finally the majority logic unit decodes the syndrome and does the error correction. The controller unit controls the error detection and correction capability of the ECC unit. The following sections describe these different components and their implementation using nano-PLA components.
3.3.1 Checker and Encoder

According to Section 3.2, each submatrix $H_i$ in Equation 1 can be expressed in the form $P_i H_1$, where $P_i$ is a permutation matrix. Figure 4 shows the modular design of the checker circuit, the hardware implementation of the parity check matrix $H$. The $i^{th}$ unit in the checker corresponds to the circuit for $H_i$, and consists of a block of XOR gates implementing $H_1$ and a reconfigurable permutation block configured to implement $P_i^T$. If the system needs to operate at an error-correcting level of $\gamma/2$, encoder units $1$ to $\gamma$ are enabled by a controller and the remaining units are disabled. The design of the checker can be implemented using the same module $H_1$ in each unit, along with a corresponding permuting array (mixer) – this makes the design regular and modular, both of which are characteristics facilitating implementation using nano-PLA components. The basic block $H_1$ is essentially a set of $n$-input XOR gates, which can be realized in nano-PLA using a nested tree configuration of the 2-input XOR gates outlined in Section 2.4. The permutation array is a reconfigurable switching circuit, which can be implemented using a nano-PLA cross-bar architecture [7].

Once the checker circuit is changed by selectively turning off some of its blocks, the corresponding encoder circuit can be modified by reconfiguring the nano-PLA implementing the encoder [7].

3.3.2 Memory

When the ECC requirements are less, the $k$ value of the code can be increased (thereby decreasing $\gamma$). As shown in Table 1, the $k$ value can be initially set to a low number (e.g., 37), so that we get high ECC in the memory. Later, as the memory ECC requirements decrease, the $k$ value can be increased to a higher number (e.g., 49). In terms of the memory, changing $k$ amounts to varying the number of information bits stored per word of memory. When $k$
becomes higher with a lower ECC requirement, more information bits can be packed into a memory code word. If the information bits are properly repacked, all the information bits in a memory bank can be stored in a lesser number of memory blocks, thereby enabling a few memory blocks to be powered off for power savings. Such a repacking architecture is proposed in Figure 5.

The repacking architecture is designed using a memory bank, which is a useful architecture in the presence of high error rates [24]. In memory banks, the read/write access to multiple memory blocks is controlled via a memory controller unit. In memory units with ECC, typically there is a scrubbing logic that periodically reads memory words, corrects them if they have errors and writes them back — this maintains the integrity of the memory. We modify the scrubbing logic to perform repacking. The repacking unit has a repacking buffer, and it uses the decoder and encoder circuits of the memory ECC. When the repacking controller receives notification from the CPU to modify the ECC and repack the memory, the repacking controller reads memory rows one by one from all the memory units in the bank except the active memory unit, from which the CPU is reading data [24] — for the active unit, its request is processed only if there is no current memory request from the CPU. The repacking controller stores the data from the rows of the different memory units into the repacking buffer, resegments the data according to the new $k$ size and writes them back to the necessary number of memory blocks. The new ECC requirement and $k$ is chosen such that the data gets repacked into fewer memory blocks, thereby making it possible to switch off one or more memory blocks. The flow of the repacking is shown in Figure 6. The example shows 120 data bits, which is first segmented using $k = 30$. After repacking, it is resegmented and

Figure 4: Modular structure of checker.
repacked with $k = 40$. This enables shutting of memory bank #4, as $n$ remains the same.

### 3.3.3 Decoder and Controller

The majority-logic (ML) decoder requires two key hardware units: XOR and majority (MAJ). A 3-input majority gate MAJ(A,B,C) can be implemented using the AND-OR planes of the nano-PLA, since MAJ(A,B,C) = AB + BC + AC. An $n$-input MAJ gate can be similarly implemented using AND/OR gates, or equivalently using the AND-OR planes of a nano-PLA. Like the encoder circuit, the decoder is also reconfigured when the ECC is modified, to handle dynamic coding. This reconfiguration can be handled easily since the MAJ decoder is implemented using reconfigurable nano-PLA.

The control signals from the memory repacking controller circuit (used to disable memory blocks in the bank architecture) are also used for selectively enabling and disabling modules of the checker, and for reconfiguring the encoder and checker components. Since we need only one controller for a complete memory bank, this module can be implemented using micro-level circuitry.

### 4 Other Properties of LDPC in Nanomemory

Apart from dynamic ECC, other properties are desirable in ECC for nanomemory designs in particular application domains. We outline two such properties.
4.1 Systematic Codes

In systematic codes, the $G$ matrix is of the form $[I : P]$, where $I$ is an identity matrix. This simplifies the encoding procedure – during encoding, the parity bits can simply be concatenated to the message bits to create a code word.

One particular type of EG-LDPC code, the two-dimensional type-I $(0, s)$-th order code [16], is a cyclic code. Any cyclic code can be converted to an equivalent systematic form [25], which implies that a 2d type-I EG-LDPC code can be expressed in systematic form. The 2d type-I EG-LDPC code has some other advantages – since it is a cyclic code, each row of the $H$ matrix can be obtained by cyclically shifting the previous row. Hence, the decoder can be efficiently implemented in hardware using a cyclic shift register [16, 2], as shown in Figure 7.

The 2d type-I EG-LDPC code can be generated easily using Euclidean Geometry. For the code with length $n = 2^{2s} - 1$, number of parity bits $n-k = 3^s-1$ and dimension $k = 2^{2s} - 3^s$, the corresponding Euclidean Geometry $EG(2, 2^s)$ consists of $2^{2s} - 1$ lines that do not pass through the origin. The $H$ matrix, which is the incidence matrix of this EG, is a $(2^{2s} - 1) \times (2^{2s} - 1)$ square matrix, where each row can be created by cyclically shifting the first row and the row weight ($\rho$) is equal to the column weight ($\gamma$). Choosing $s = 3$, we get $n = 63$, $k = 37$ and $\rho = \gamma = 16$, which means that this code can be used in a 64-bit

![Figure 6: Flow of memory repacking.](image)
Figure 7: Cyclic shift register-based decoder for EG-LDPC type-I code.

memory architecture (one additional bit can be packed with an overall parity) requiring systematic encoding and capacity of correcting 8 errors per word.

4.2 Modular Codes

A desirable property in some nanomemory applications is modularity of hardware design of encoder and decoder. This makes the hardware design quite simple, since one can optimize the design of a single module and use multiple such modules to create both the encoder and the decoder circuits.

We will show how a particular class of EG-LDPC code, type-II, has this desirable modularity property. Type-II EG-LDPC codes are quasi-cyclic, i.e., each row of their $H$-matrix can be obtained by shifting previous rows by a particular fixed number of positions $c$ (if $c = 1$, the code is cyclic $\Rightarrow$ cyclic codes are a special case of quasi-cyclic codes). Now, any quasi-cyclic matrix can be equivalently put in a circulant form, where the $H$-matrix comprises a set of circulants [15]. Note that a circulant is a square matrix where each row is a cyclic shift of the row above it (with wraparound of the top row), and each column is a cyclic shift of the column to its left (with wraparound of the leftmost row). A detailed example of quasi-cyclic codes, circulants, and the equivalence between these two is shown in Figure 8. Recent work has shown that the encoder matrix can also be obtained in modular form for certain specific kinds of type-II codes [11].
5 Analysis and Experiments

We analyze the effects of faults and dynamic coding in the various components of the model outlined in Section 3.3.

5.1 Fault Tolerance

Let $\epsilon_e$, $\epsilon_m$ and $\epsilon_c$ be random variables signifying the number of errors in the encoder, memory, and checker blocks, respectively. According to Section 3.3.3, when $\gamma$ components are enabled by the controller, the overall error correcting capacity of the ECC memory system (with ML decoding) is $\gamma/2$. Therefore, the majority-logic decoder will be able to correct errors as long as

$$\epsilon_e + \epsilon_m + \epsilon_c \leq \gamma/2.$$  \hspace{1cm} (2)

Note that the MAJ decoder is assumed to be fault free, which can be ensured by using self-checking in the MAJ logic.

Let us now analyze each component of the above equation in detail. When $\gamma$ units are enabled by the controller, the code is a $(\gamma, \rho)$-regular Gallager code according to Section 3.2, i.e., the check matrix unit $H_1$ has a single 1 in each column and $\rho$ 1’s in each row. Consequently, each XOR gate realizing the $H_1$ matrices are $\rho$-input XOR gates. In this analysis, we will consider the effects
of errors in the XOR gates to be additive. In some cases errors can mask each other, e.g., complementary soft errors in two inputs of an XOR gate can mask each other – we do not consider that in this worst-case analysis.

Let $p_e$ and $p_c$ be the errors in one bit location of an XOR gate in the encoder and checker, respectively, and $p_m$ be the probability of a nano-PLA memory junction losing its charge. Assuming errors to be i.i.d. and the total number of code bits to be $n$, we get

$$\text{Prob}(e \text{ out of } n \text{ code bits have errors}) = \binom{n}{e} (p)^e (1 - p)^{n-e}, \quad (3)$$

where putting $p = p_j$ gives us the error probability in the memory, while selecting $p = p_c$ gives us the error probability in the encoder or checker. Considering Equations (2) and (3), we see that the distribution of errors in the overall ECC memory system is the sum of binomial random variables. Therefore, the probability of the majority decoder not having any error is given by the cumulative distribution of the convolution of binomial distributions.

Figure 9 shows how the probability of error-free operation of a memory system with EG-LDPC error correction changes along with different values of $\gamma$ (the number of stages in the dynamic error control), for $p_e, p_c = 1\%$ (error rates for encoder and checker) and $p_m = 2\%$ (error rate for memory). As shown in the figure, for an 8-bit memory, $\gamma = 6$ gives almost 100% probability of correct operation; the same high reliability is obtained using $\gamma = 8$ for a 16-bit memory and $\gamma = 12$ for a 32-bit memory. Figure 10 shows the plot for a 64-bit memory (corresponding to $m = 6$), for a higher error rate (8%) — using $\gamma = 32$ in this case gives $> 99\%$ probability of correct operation. Note that
using Gallager’s bit-flipping algorithm (as discussed in Section 3.2) can give us similar probability of correct operation with lesser \( \gamma \) (and hence a lesser number of hardware component modules), at the cost of more complex hardware in each module. Note that in all these figures, the size of the memory is \( n = 2^m \), for a \((m, 2^s)\) EG-LDPC code.

5.2 Dynamic Coding

Along the bathtub curve, the probability of failure of components varies — high in the beginning, low during normal operations, and finally high again. Figure 11 shows how the \( \gamma \) value required (for > 99% probability of normal operation) varies with changing probability of failure of each component (in this case, we assume \( m = 5 \) and \( p = p_c = p_d = p_m \)). When \( p = 0.06 \), the required \( \gamma = 15 \), but as \( p \) decreases to 0.01, \( \gamma \) decreases steadily to 5.

Figure 12 shows a simulation of the bathtub curve of failure — initially, during infant mortality failure, a high value of \( \gamma \) is required. During the middle phase, when there is a low constant failure rate, the \( \gamma \) required is less, so that some of the components of the checker and memory bank can be disabled, leading to savings in static power dissipation. Toward the end of the device lifetime, when wearout failures increase the error rate, the required \( \gamma \) increases again, leading to increased power dissipation. This demonstrates that for lower probability of component failure, we can selectively disable the blocks in the checker and memory bank and obtain substantial power savings. Thus, dynamic error correction is well suited in this domain.
Figure 11: Change of $\gamma$ (for correct operation of the ECC memory with > 99% probability) with different values of $p$.

Figure 12: Percentage of maximum power dissipated (for correct operation of the ECC memory with > 99% probability) along the bathtub curve of failure.
6 Area Overhead

The different components of the memory LDPC-ECC system implemented using nanocomponents are (1) encoder, (2) memory, (3) checker, and (4) decoder. Note that the controller is designed using micro-level circuitry for the whole memory bank; hence, we do not consider its area in our current analysis.

An upper bound on the number of 2-input XOR gates required for the encoder and checker circuits is $(n - k)(k - 1) + J(\rho - 1)$. The decoder has $n \gamma$-input MAJ gates, which can be synthesized using 2-input AND/OR gates or directly with a nano-PLA. In either case, each of the $n \gamma$-input MAJ gates will need on the order of $(2^\gamma - 1)$ 2-input AND gates and $(\gamma - 1)$ 2-input OR gates. Let us consider that a memory unit is composed of 6 nanotransistors, while 2-input OR, AND, and XOR gates are composed of 4, 4, and 8 nanotransistors respectively [20]. For a 2-dimensional $(64, 40)$ EG-Gallager code with $\gamma = 6, \rho = 6, J = 64$, constructed based on the Euclidean Geometry $EG(2, 2^3)$, the area of the encoder, checker and decoder circuits is 44K transistors. The corresponding area for a 64-bit memory with 1K rows is 384K transistors. Table 2 shows the area overhead of having LDPC-ECC in the memory, where area is measured in terms of number of transistors. As the size of the memory block increases, the overhead of the parity bits and the ECC logic converges to $(1 - \text{coderate}) = (n - k)/n = 37.5\%$, which indicates that for large memory blocks the overhead due to the ECC encoder, checker, and decoder circuits is very low (e.g., for 16K memory, it is 0.5\%). This is consistent with observations by other researchers on other types of LDPC codes [20].

7 Irregular LDPC Codes

There are different types of LDPC codes where the number of 1’s in the rows and columns are not uniform — some of these irregular codes have the properties of modular, systematic, and dynamic codes that we have been studying in this paper. For example, Accumulate-Repeat-Accumulate (ARA) codes [2], which are constructed by suitably repeating core protograph structures, can have both modular encoder and decoder matrices. The same property holds for Cycle-invariant Difference Set (CIDS) codes, which are generated using difference sets over Abelian groups and have block-circulant structure of both encoder and
decoder matrices [18]. Both these codes can be used as dynamic codes — however, they are not well suited for our nano-PLA-based architecture, since these codes are not MAJ-logic decodable and need belief propagation-based decoders, which are significantly more complex to implement in nanohardware.

8 Related Work

LDPC codes have been widely used in different communication applications [16], especially after Mackay’s recent work revived interest in them [17]. In our work, we use a property of EG-LDPC codes that allows simpler decoding logic (involving majority gates) to be used, which can be implemented in nanohardware using different techniques [3]. LDPC codes defined over other finite geometries have been studied lately [26], and hardware for performing efficient decoding of LDPC codes has been proposed [6, 19].

The work in this paper is focused on ECC for nanomemory. Other error-correcting codes have been used at nanoscale — recently a hierarchical fault tolerance technique, using Hamming codes, was proposed for nanocomputing operations [23].

One aspect of our current work is that the ECC circuit components themselves can have faults in them, which must be accounted for during error correction. Another scheme for handling faulty hardware modules was considered in von Neumann’s seminal fault-tolerant multiplexing scheme [29], which has also been extended recently using a probabilistic computation model with noisy gates [22]. In micro-level circuits, various other fault tolerance techniques have been used, e.g., total self-checking [1, 27], parity prediction [28].

9 Conclusions and Future Work

We want to focus on several interesting extensions to this work. If there are line-level faults in the address lines or row/column lines, then it is useful to have block-level error correcting codes like Reed-Solomon [16] instead of LDPC. If the whole memory block becomes unusable because of a fault, then it is useful to have distributed error correction in different memory blocks instead of a centralized error-correcting scheme. To that effect, we intend to explore striped RAID-type architecture for nanomemory banks [14]. In the nanodomain, asymmetric and unidirectional faults might occur — we plan to investigate such error models, study the prevalence of these types of errors through fault simulations in models of nanocomputing, and handle such faults using corresponding error-correcting codes, e.g., asymmetric codes, unidirectional codes [4]. Finally, we want to implement an actual prototype of the proposed architecture on the nano-PLA substrate and perform experiments with actual memory traces from different domains [13], injecting faults following different fault models.
References


