Fault Secure Encoder and Decoder for NanoMemory Applications

Helia Naeimi and André DeHon

***Abstract*—Memory cells have been protected from soft errors for more than a decade; due to the increase in soft error rate in logic circuits, the encoder and decoder circuitry around the memory blocks have become susceptible to soft errors as well and must also be protected. We introduce a new approach to design fault-secure encoder and decoder circuitry for memory designs. The key novel contribution of this paper is identifying and defining a new class of error-correcting codes whose redundancy makes the design of fault-secure detectors (FSD) particularly simple. We further quantify the importance of protecting encoder and decoder circuitry against transient errors, illustrating a scenario where the system failure rate (FIT) is dominated by the failure rate of the encoder and decoder. We prove that *Euclidean Geometry Low-Density Parity-Check* (EG-LDPC) codes have the fault-secure detector capability. Using some of the smaller EG-LDPC codes, we can tolerate bit or nanowire defect rates of 10% and fault rates of 10^{-18} upsets/device/cycle, achieving a FIT rate at or below one for the entire memory system and a memory density of 10^{11} bit/cm^2 with nanowire pitch of 10 nm for memory blocks of 10 Mb or larger. Larger EG-LDPC codes can achieve even higher reliability and lower area overhead.**

***Index Terms*—Decoder, encoder, fault tolerant, memory, nanotechnology.**

I. Introduction and Motivation

NANOTECHNOLOGY provides smaller, faster, and lower energy devices which allow more powerful and compact circuitry; however, these benefits come with a cost—the nanoscale devices may be less reliable. Thermal- and shot-noise estimations [8], [12] alone suggest that the transient fault rate of an individual nanoscale device (e.g., transistor or nanowire) may be orders of magnitude higher than today's devices. As a result, we can expect combinational logic to be susceptible to transient faults in addition to storage cells and communication channels. Therefore, the paradigm of protecting only memory cells and assuming the surrounding circuitries (i.e., encoder and decoder) will never introduce errors is no longer valid.

Manuscript received January 29, 2008; revised July 14, 2008. First published February 27, 2009; current version published March 18, 2009. This research was supported in part by National Science Foundation Grant CCF-0403674 and by the Defense Advanced Research Projects Agency under ONR Contract N00014-01-0651.

H. Naeimi was with the Computer Science Department, California Institute of Technology, Pasadena, CA 91125 USA. She is now with Intel Research, Santa Clara Laboratory, Santa Clara, CA 95054 USA (e-mail: helia.naeimi@intel.com).

A. DeHon is with the Department of Electrical and Systems Engineering, University of Pennsylvania, Philadelphia, PA 19104 USA (e-mail: andre@acm.org).

Digital Object Identifier 10.1109/TVLSI.2008.2009217

In this paper, we introduce *a fault-tolerant nanoscale memory architecture which tolerates transient faults* **both** *in the storage unit and in the supporting logic* (i.e., encoder, decoder (corrector), and detector circuitries).

Particularly, we identify a class of error-correcting codes (ECCs) that guarantees the existence of a simple fault-tolerant detector design. This class satisfies a new, restricted definition for ECCs which guarantees that the ECC codeword has an appropriate redundancy structure such that it can detect multiple errors occurring in both the stored codeword in memory and the surrounding circuitries. We call this type of error-correcting codes, *fault-secure detector capable ECCs* (FSD-ECC). The parity-check Matrix of an FSD-ECC has a particular structure that the decoder circuit, generated from the parity-check Matrix, is Fault-Secure. The ECCs we identify in this class are close to optimal in rate and distance, suggesting we can achieve this property without sacrificing traditional ECC metrics.

We use the fault-secure detection unit to design a fault-tolerant encoder and corrector by monitoring their outputs. If a detector detects an error in either of these units, that unit must repeat the operation to generate the correct output vector. Using this retry technique, we can correct potential transient errors in the encoder and corrector outputs and provide a fully fault-tolerant memory system.

The novel contributions of this paper include the following:

1) a mathematical definition of ECCs which have simple FSD which do not requiring the addition of further redundancies in order to achieve the fault-secure property (see Section IV);
2) identification and proof that an existing LDPC code (EG-LDPC) has the FSD property (see Section V);
3) a detailed ECC encoder, decoder, and corrector design that can be built out of fault-prone circuits when protected by this fault-secure detector also implemented in fault-prone circuits and guarded with a simple OR gate built out of reliable circuitry (see Section VI).

To further show the practical viability of these codes, we work through the engineering design of a nanoscale memory system based on these encoders and decoders including the following:

- memory banking strategies and scrubbing (see Section VI-E);
- reliability analysis (see Section VII);
- unified ECC scheme for both permanent memory bit defects and transient upsets (see Section VIII).

This allows us to report the area, performance, and reliability achieved for systems based on these encoders and decoders (see Section IX). We start by reviewing the related work (see Section II) to put this development in context, and we provide

an overview of the system architecture for the fault-tolerant memory system (see Section III) which uses these circuits.

We first introduced the concept of ECC with fault-secure detector in [19]; then we developed the nanowire-based implementation and memory-system optimization in [20]. In addition to combining the results from our earlier papers, this paper describes how to use the single ECC system for both permanent defect and transient fault tolerance. This article also provides a more detailed accounting of net memory bit density.

II. Related Work

Traditionally, memory cells were the only circuitry susceptible to transient faults, and all the supporting circuitries around the memory (i.e., encoders and decoders) were assumed to be fault-free. As a result most of prior work designs for fault-tolerant memory systems focused on protecting only the memory cells. However, as we continue scaling down feature sizes or use sublithographic devices, the surrounding circuitries of the memory system will also be susceptible to permanent defects and transient faults [11].

One approach to avoid the reliability problem in the surrounding circuitries is to implement these units with more reliable devices (e.g., more reliable CMOS technologies [6], [26]). However, from an area, performance, and power consumption point of view it is beneficial to implement encoders and decoders with scaled feature size or nanotechnology devices. Consequently, it is important to remove the reliability barrier for these logic circuits so they can be implemented with scaled feature size or nanotechnology devices.

Almost all of the proposed fault tolerant encoders and decoders so far, use the conventional fault tolerant scheme (e.g., logic replication or concurrent parity prediction) to protect the encoder and corrector circuitry. That is, they add additional logic to check the correctness of the circuit calculation. In contrast, the technique introduced in this work exploits the existing structure of the ECC to guarantee the fault-secure property of the detector unit *without adding redundant computations*.

The work presented in [21], is an example of the scheme using redundancy to generate fault tolerant encoder. Reference [21] develops a fault-secure encoder unit using a concurrent parity-prediction scheme. Like the general parity-prediction technique, [21] concurrently generates (predicts) the parity-bits of the encoder outputs (encoded bits) from the encoder inputs (information bits). The predicted parity bits are then compared against the actual parity function of the encoder output (encoded bits) to check the correctness of the encoder unit. The parity predictor circuit implementation is further optimized for each ECC to make a more compact design. For this reason, efficient parity-prediction designs are tailored to a specific code. Simple parity prediction guarantees single error detection; however, no generalization is given for detecting multiple errors in the detector other than complete replication of the prediction and comparison units.

In contrast, our design detects multiple errors in the encoder and corrector units. Furthermore, the fault-secure detector and checking logic for our FSD-ECC codes can be automatically generated from the already known parity-check matrix. This fault secure detector delivers protection without requiring additional fault-tolerance circuitry.

III. System Overview

In this section, we outline our memory system design that can tolerate errors in any part of the system, including the storage unit and encoder and corrector circuits using the fault-secure detector. For a particular ECC used for memory protection, let E be the maximum number of error bits that the code can correct and D be the maximum number of error bits that it can detect, and in one error combination that strikes the system, let e_e, e_m, and e_c be the number of errors in encoder, a memory word, and corrector, and let e_{de} and e_{dc} be the number of errors in the two separate detectors monitoring the encoder and corrector units. In conventional designs, the system would guarantee error correction as long as $e_m \leq E$ and $e_e = e_c = 0$. In contrast, here we guarantee that the system can correct any error combination as long as $e_m \leq E$, $e_e + e_{de} \leq D$, and $e_m + e_c + e_{dc} \leq D$. This design is feasible when the following two fundamental properties are satisfied:

1) any single error in the encoder or corrector circuitry can at most corrupt a single codeword bit (i.e., no single error can propagate to multiple codeword bits);
2) there is a fault secure detector that can detect any combination of errors in the received codeword along with errors in the detector circuit. This fault-secure detector can verify the correctness of the encoder and corrector operation.

The first property is easily satisfied by preventing logic sharing between the circuits producing each codeword bit or information bit in the encoder and the corrector respectively. In Section IV, we define the requirements for a code to satisfy the second property.

An overview of our proposed reliable memory system is shown in Fig. 1 and is described in the following. The information bits are fed into the encoder to encode the information vector, and the fault secure detector of the encoder verifies the validity of the encoded vector. If the detector detects any error, the encoding operation must be redone to generate the correct codeword. The codeword is then stored in the memory. During memory access operation, the stored codewords will be accessed from the memory unit. Codewords are susceptible to transient faults while they are stored in the memory; therefore a corrector unit is designed to correct potential errors in the retrieved codewords. In our design (see Fig. 1) all the memory words pass through the corrector and any potential error in the memory words will be corrected. Similar to the encoder unit, a fault-secure detector monitors the operation of the corrector unit. All the units shown in Fig. 1 are implemented in fault-prone, nanoscale circuitry; the only component which must be implemented in reliable circuitry are two OR gates that accumulate the syndrome bits for the detectors (shown in Fig. 3 and described in Section VI-B).

Data bits stay in memory for a number of cycles and, during this period, each memory bit can be upset by a transient fault with certain probability. Therefore, transient errors accumulate in the memory words over time. In order to avoid accumulation of too many errors in any memory word that surpasses the code

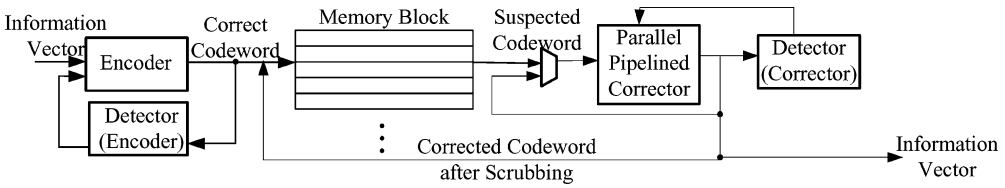

Fig. 1. Overview of our proposed fault-tolerant memory architecture, with pipelined corrector.

correction capability, the system must perform memory *scrubbing*. Memory scrubbing is the process of periodically reading memory words from the memory, correcting any potential errors, and writing them back into the memory (e.g., [22]). This feature is shown in the revised system overview in Fig. 10. To perform the periodic scrubbing operation, the normal memory access operation is stopped and the memory performs the scrub operation. In Section VI, we further detail the scrubbing operation and potential optimization to achieve high performance and high reliability. We also explain each of the above units, encoder, corrector, and detector, and memory operations in greater detail in Section VI.

IV. ECCs With Fault Secure Detector

In this section, we present our novel, restricted ECC definition for our fault-secure detector capable codes. Before starting the details of our new definition we briefly review basic linear ECCs.

A. Error-Correcting Code Reviews

Let $i = (i_0, i_1, \ldots, i_{k-1})$ be the k-bit information vector that will be encoded into an n-bit codeword, $c = (c_0, c_1, \ldots, c_{n-1})$. For linear codes, the encoding operation essentially performs the following vector-matrix multiplication:

$$c = i \cdot G \tag{1}$$

where G is a $k \times n$ generator matrix. The validity of a received encoded vector can be checked with the *Parity-Check* matrix, which is an $(n-k) \times n$ binary matrix named H. The checking or detecting operation is basically summarized as the following vector-matrix multiplication:

$$s = c \cdot H^T. \tag{2}$$

The $(n-k)$-bit vector s is called the *syndrome* vector. A syndrome vector is zero if c is a valid codeword, and nonzero if c is an erroneous codeword. Each code is uniquely specified by its generator matrix or parity-check matrix.

A code is a *systematic code* if every codeword consists of the original k-bit information vector followed by $n-k$ parity bits [16]. With this definition, the generator matrix of a systematic code must have the following structure:

$$G = [I : X] \tag{3}$$

where I is a $k \times k$ identity matrix and X is a $k \times (n-k)$ matrix that generates the parity-bits. The advantage of using systematic codes is that there is no need for a decoder circuit to extract the information bits. The information bits are simply available in the first k bits of any encoded vector.

A code is said to be a *cyclic code* if for any codeword c, all the cyclic shifts of the codeword are still valid codewords. A code is cyclic *iff* the rows of its parity-check matrix and generator matrix are the cyclic shifts of their first rows.

The *minimum distance* of an ECC, d, is the minimum number of code bits that are different between any two codewords. The maximum number of errors that an ECC can detect is $d-1$, and the maximum number that it corrects is $\lfloor d/2 \rfloor$. Any ECC is represented with a triple (n, k, d), representing code length, information bit length, and minimum distance, respectively.

B. FSD-ECC Definition

The restricted ECC definition which guarantees a FSD-ECC is as follows.

Definition I: Let C be an ECC with minimum distance d. C is FSD-ECC if it can detect any combination of overall $d-1$ or fewer errors in the received codeword and in the detector circuitry.

Theorem I: Let C be an ECC, with minimum distance d. C is FSD-ECC *iff* any error vector of weight $0 < e \leq d-1$, has syndrome vector of weight at least $d-e$.

Note: The following proof depends on the fact that any single error in the detector circuitry can corrupt at most one output (one syndrome bit). This can be easily satisfied for any circuit by implementing the circuit in such a way that no logic element is shared among multiple output bits; therefore, any single error in the circuit corrupts at most one output (one syndrome bit).

Proof: The core of a detector circuitry is a multiplier that implements the vector-matrix multiply of the received vector and the parity-check matrix to generate the syndrome vector. Now if e errors strike the received codeword the syndrome weight of the error pattern is at least $d-e$ from the assumption. Furthermore, the maximum number of tolerable errors in the whole system is $d-1$ and e errors already exist in the encoded vector, therefore the maximum number of errors that can strike in the detector circuitry is $d-1-e$. From the previous note, these many errors can corrupt at most $d-1-e$ syndrome bit, which in worst case leaves at least one nonzero syndrome bit and therefore detects the errors. **Q.E.D.**

The difference between FSD-ECC and normal ECC is simply the demand on syndrome weight. That is, for error vector of weight $e > 0$, a normal ECC demands nonzero syndrome weight while FSD-ECC demands syndrome weight of $\geq d-e$.

V. FSD-ECC Example: Euclidean Geometry Codes

A. Euclidean Geometry Code Review

This section reviews the construction of Euclidean Geometry codes based on the lines and points of the corresponding finite geometries [27]. Euclidean Geometry codes are also called EG-LDPC codes based on the fact that they are low-density parity-check (LDPC) codes [14]. LDPC codes have a limited number of 1's in each row and column of the matrix; this limit guarantees limited complexity in their associated detectors and correctors making them fast and light weight [9].

Let $\mathbf{EG}$ be a Euclidean Geometry with n points and J lines. $\mathbf{EG}$ is a finite geometry that is shown to have the following fundamental structural properties:

1) every line consists of ρ points;
2) any two points are connected by exactly one line;
3) every point is intersected by γ lines;
4) two lines intersect in exactly one point or they are parallel; i.e., they do not intersect.

Let H be a $J \times n$ binary matrix, whose rows and columns corresponds to lines and points in an $\mathbf{EG}$ Euclidean geometry, respectively, where $h_{i,j} = 1$ if and only if the ith line of $\mathbf{EG}$ contains the jth point of $\mathbf{EG}$, and $h_{i,j} = 0$ otherwise. A row in H displays the points on a specific line of $\mathbf{EG}$ and has weight ρ. A column in H displays the lines that intersect at a specific point in $\mathbf{EG}$ and has weight γ. The rows of H are called the incidence vectors of the lines in $\mathbf{EG}$, and the columns of H are called the intersecting vectors of the points in $\mathbf{EG}$. Therefore, H is the incidence matrix of the lines in $\mathbf{EG}$ over the points in $\mathbf{EG}$. It is shown in [15] that H is a LDPC matrix, and therefore the code is an LDPC code.

A special subclass of EG-LDPC codes, type-I 2-D EG-LDPC, is considered here. It is shown in [15] that type-I 2-D EG-LDPC have the following parameters for any positive integer $t \geq 2$:

- information bits, $k = 2^{2t} - 3^t$;
- length, $n = 2^{2t} - 1$;
- minimum distance, $d_{\min} = 2^t + 1$;
- dimensions of the parity-check matrix, $n \times n$;
- row weight of the parity-check matrix, $\rho = 2^t$;
- column weight of the parity-check matrix, $\gamma = 2^t$.

It is important to note that the rows of H are not necessarily linearly independent, and therefore the number of rows do not necessarily represents the rank of the H matrix. The rank of H is $n-k$ which makes the code of this matrix (n, k) linear code. Since the matrix is $n \times n$, the implementation has n syndrome bits instead of $n-k$. The $(2^{2t}-1)\times(2^{2t}-1)$, parity-check matrix H of an $\mathbf{EG}$ euclidean geometry, can be formed by taking the incidence vector of a line in $\mathbf{EG}$ and its $2^{2t}-2$ cyclic shifts as rows; therefore this code is a *cyclic code*.

B. FSD-ECC Proof for EG-LDPC

In this section, we prove that EG-LDPC codes have the FSD-ECC property.

Theorem II: Type-I 2-D EG-LDPC codes are FSD-ECC.

Proof: Let C be an EG-LDPC code with column weight γ and minimum distance d. We have to show that any error vector of weight $0 < e \leq d-1$ corrupting the received encoded vector has syndrome vector of weight at least $d - e$.

TABLE I
EG-LDPCs and Upper and Lower Bounds on Code Length

Hamming bound	EG-LDPC	Gilbert-Varshamov bound
(14,7,5)	(15,7,5)	(17,7,5)
(58,37,9)	(63,37,9)	(67,37,9)
(222,175,17)	(255,175,17)	(255,175,17)

Now a specific bit in the syndrome vector will be one if and only if the parity-check sum corresponding to this syndrome vector has an odd number of error bits present in it. Looking from the Euclidean geometry perspective, each error bit corresponds to a point in the geometry and each bit in the syndrome vector corresponds to a line. Consequently, we are interested in obtaining a lower bound on the number of lines that pass through an odd number of error points. We further lower bound this quantity by the number of lines that pass through exactly one of the error points. Based on the definition of the Euclidean geometry, γ lines pass through each point; so e error points potentially impact γe lines. Also at most one line connects two points. Therefore, looking at the e error points, there are at most $\binom{e}{2}$ lines between pairs of error points. Hence, the number of lines passing through a collection of these e points is lower bounded by $\gamma e - \binom{e}{2}$. Out of this number, at most $\binom{e}{2}$ lines connect two or more points of the error points. Summarizing all this, the number of lines passing through exactly one error point, which gives us the lower bound on the syndrome vector weight, is at least $\gamma e - 2\binom{e}{2}$.

From the code properties introduced in Section V-A and knowing that $d = \gamma + 1$, we can derive the following inequality:

$$|s(c_e)| \geq \gamma e - 2\binom{e}{2} = e(\gamma + 1 - e) = e(d - e)$$
$$\geq d - e \quad \text{when} \quad e > 0$$

The previous inequality says that the weight of the syndrome vector of a codeword with e errors is at least $d - e$ when $e > 0$ which is the required condition of Theorem (I). Therefore, EG-LDPC is FSD-ECC. **Q.E.D.**

C. Efficiency of EG-LDPC

It is important to compare the rate of the EG-LDPC code with other codes to understand if the interesting properties of low-density and FSD-ECC come at the expense of lower code rates. We compare the code rates of the EG-LDPC codes that we use here with an achievable code rate upper bound (*Gilbert-Varshamov bound*) and a lower bound (*Hamming bound*). Table I shows the upper and lower bounds on the code overhead, for each of the used EG-LDPC. The EG-LDPC codes are no larger than the achievable Gilbert bound for the same k and d value, and they are not much larger than the Hamming bounds. Consequently, we see that we achieve the FSD property without sacrificing code compactness.

VI. Design Structure

In this section, we provide the design structure of the encoder, corrector, and detector units of our proposed fault-tolerant memory system. We also present the implementation of

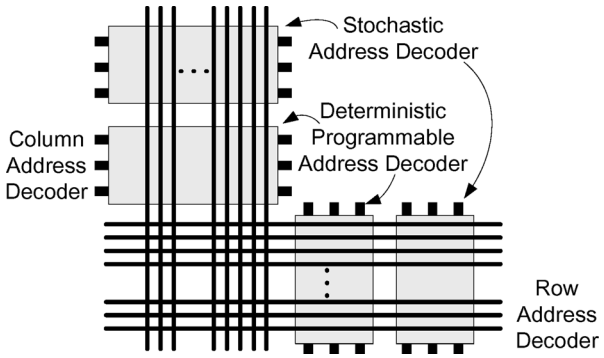

Fig. 2. Structure of NanoMemory core. Please note that the address decoders are for memory access and are not related to the memory ECC decoders.

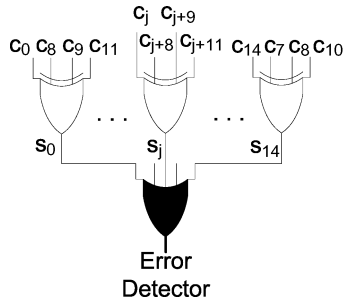

Fig. 3. Fault-secure detector for (15, 7, 5) EG-LDPC code. All the gates except the last OR gate are implemeneted with fault-prone nanoscale circuitry. The last OR gate is implemented with more reliable lithography technique.

TABLE II
DETECTOR, ENCODER, AND CORRECTOR CIRCUIT AREA IN THE NUMBER OF 2-INPUT GATES

Code	(15,7,5)	(63,37,9)	(255,175,17)
Detector	45	501	3825
Encoder	22	355	6577
Serial Corrector	19	83	331
Parallel Corrector	285	5229	84405

these units on a sub-lithographic, nanowire-based substrate. Before going into the design structure details we start with a brief overview of the sub-lithographic memory architecture model.

A. NanoMemory Architecture Model

We use the *NanoMemory* [4], [6] and *NanoPLA* [5] architectures to implement the memory core and the supporting logic, respectively. NanoMemory and NanoPLA are based on nanowire crossbars [2], [10].

The NanoMemory architecture developed in [4], [6] can achieve greater than 10^{11} b/cm^2 density even after including the lithographic-scale address wires and defects. This design uses a nanowire crossbar to store memory bits and a limited number of lithographic scale wires for address and control lines. Fig. 2 shows a schematic overview of this memory structure. The fine crossbar shown in the center of the picture stores one memory bit in each crossbar junction. To be able to write the value of each bit into a junction, the two nanowires crossing that junction must be uniquely selected and an adequate voltage must be applied to them (e.g., [3], [24]). The nanowires can be uniquely selected through the two address decoders located on the two sides of the memory core.

The detail of the NanoMemory structure is presented in [4] and [6]. For our design, we revise the original NanoMemory structure introduced in [4] and [6]. Instead of using a lithographic-scale interface to read and write into the memory core, we use a nanowire-based interface. The reason that we can remove the lithographic-scale interface is that all the blocks interfacing with the memory core (encoder, corrector and detectors) are implemented with nanowire-based crossbars. So we use a nanowire-based DEMUX to connect the memory core to the supporting logic blocks. The detail of the DEMUX structure is available in [17], [18], and [20].

B. Fault Secure Detector

The core of the detector operation is to generate the syndrome vector, which is basically implementing the following vector-matrix multiplication on the received encoded vector c and parity-check matrix H:

$$s = c \cdot H^T. \tag{4}$$

Therefore each bit of the syndrome vector is the product of c with one row of the parity-check matrix. This product is a linear binary sum over digits of c where the corresponding digit in the matrix row is 1. This binary sum is implemented with an XOR gate. Fig. 3 shows the detector circuit for the (15, 7, 5) EG-LDPC code. Since the row weight of the parity-check matrix is ρ, to generate one digit of the syndrome vector we need a ρ-input XOR gate, or $(\rho-1)$ 2-input XOR gates. For the whole detector, it takes $n(\rho-1)$ 2-input XOR gates. Table II illustrates this quantity for some of the smaller EG-LDPC codes. Note that implementing each syndrome bit with a separate XOR gate satisfies the assumption of Theorem I of no logic sharing in detector circuit implementation.

An error is detected if any of the syndrome bits has a nonzero value. The final error detection signal is implemented by an OR function of all the syndrome bits. The output of this n-input OR gate is the error detector signal (see Fig. 3). In order to avoid a single point of failure, we must implement the OR gate with a reliable substrate (e.g., in a system with sub-lithographic *nanowire* substrate, the OR gate is implemented with reliable lithographic technology—i.e., lithographic-scaled wire-OR). This n-input wire-OR is much smaller than implementing the entire $n \times (\rho-1)$ 2-input XORs at the lithographic scale. The area of each detector is computed using the model of NanoPLA and NanoMemory form [5] and [4] and [6] accounting for all the supporting lithographic wires and reported in Table III.

Fig. 4 shows the implementation of the detector on a NanoPLA substrate. The framed block in Fig. 4(b) shows a ρ-input XOR gate, implementing a $\log_2(\rho)$-level XOR tree in spiral form (see Fig. 4(a)). The solid boxes display the restoration planes and the white boxes display the wired-OR planes of NanoPLA architecture model [5], [7]. In Fig. 4, the signals rotate counter clock-wise, and each round of signal generates the XOR functions of one level of the XOR-tree. The final output then gates a robust lithographic-scale wire. This lithographic-scale wire generates a wired-OR function of all the

TABLE III
DECOMPOSED AREA PER BIT OF THE DESIGN POINTS SELECTED FOR THE CURVES IN Fig. 14. THE MEMORY SIZE FOR THIS DESIGN IS 10^{12} bit. THE UNIT OF AREA IS NANOMETERS SQUARED PER BIT

Code (n,k,d)	Unprotected Memory	Protected Memory	Global Enc.+Det.	Cluster Cor.	Cluster Det.	C	D_{thr}	S (min)	Thr Loss	log(FIT)
(15,7,5)	626	1050	3.15E-5	0.180	0.03	100	1	10	0.10%	-1.9
(63,37,9)	523	719	7.20E-5	0.273	0.02	1000	2	120	0.08%	-9.0
(63,37,9)	2050	2817	7.20E-5	0.273	0.02	1000	1	120	0.08%	-23
(255,175,17)	521	746	8.65E-3	7.36	0.15	1000	4	120	0.08%	-35
(255,175,17)	745	1066	8.65E-3	7.36	0.15	1000	3	120	0.08%	-49
(255,175,17)	1524	2180	8.65E-3	7.36	0.15	1000	2	120	0.08%	-64

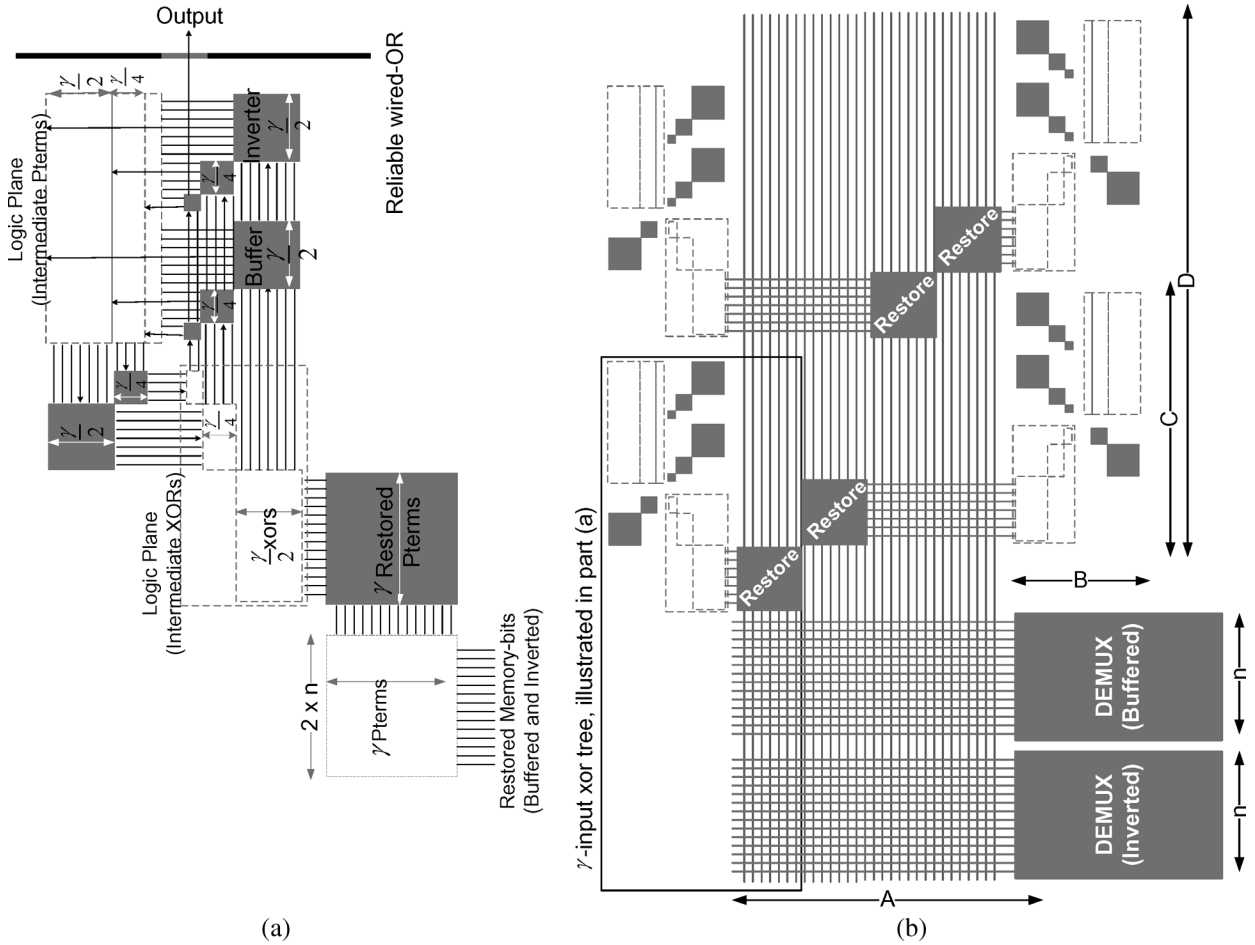

Fig. 4. (a) γ-input XOR tree implemented on NanoPLA structure. (b) Detector circuit implemented on NanoPLA: The parameters in the figure are $A = n \times \gamma$, $B = 2 \times Pl(\gamma - 1)$, $C = Pl(2 \times \gamma - 2) + 2 \times Pl(\gamma - 1)$, and $D = n/2 \times C$, where $Pl(x)$, is the width of a NanoPLA plane with x nanowire, including the area of the supporting lithographic scale wires.

n ρ-input XORs and is the final output of the detector circuit. The XOR gate is the main building block of the encoder and corrector as well.

C. Encoder

An n-bit codeword c, which encodes a k-bit information vector i is generated by multiplying the k-bit information vector with a $k \times n$ bit generator matrix G; i.e., $c = i \cdot G$.

EG-LDPC codes are not systematic and the information bits must be decoded from the encoded vector, which is not desirable for our fault-tolerant approach due to the further complication and delay that it adds to the operation. However, these codes are cyclic codes [15]. We used the procedure presented in [15] and [16] to convert the cyclic generator matrices to systematic generator matrices for all the EG-LDPC codes under consideration.

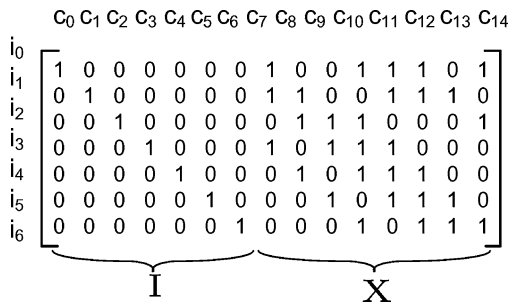

Fig. 5. Generator matrix for the (15, 7, 5) EG-LDPC in systematic format; note the identity matrix in the left columns.

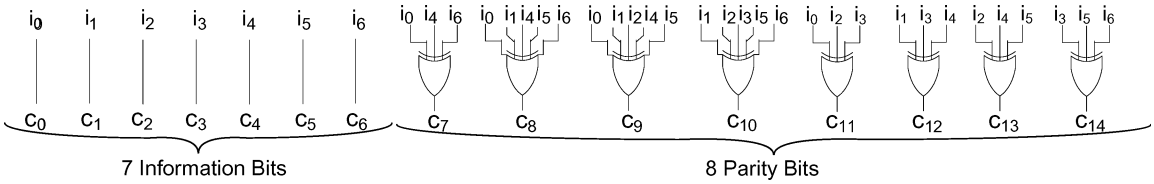

Fig. 6. Structure of an encoder circuit for the (15, 7, 5) EG-LDPC code; i_0 to i_6 are 7-bit information vector. Each of the XOR gates generate one parity bit of the encoded vector. The codeword consists of seven information bits followed by eight parity bits.

Fig. 5 shows the systematic generator matrix to generate (15, 7, 5) EG-LDPC code. The encoded vector consists of information bits followed by parity bits, where each parity bit is simply an inner product of information vector and a column of X, from $G = [I : X]$. Fig. 6 shows the encoder circuit to compute the parity bits of the (15, 7, 5) EG-LDPC code. In this figure $i = (i_0, \ldots, i_6)$ is the information vector and will be copied to $c_0, \ldots, c_6$ bits of the encoded vector, c, and the rest of encoded vector, the parity bits, are linear sums (XOR) of the information bits.

If the building block is two-input gates then the encoder circuitry takes 22 two-input XOR gates. Table II shows the area of the encoder circuits for each EG-LDPC codes under consideration based on their generator matrices.

Once the XOR functions are known, the encoder structure is very similar to the detector structure shown in Fig. 4, except it consists of $(n-k)$ XOR gates of varying numbers of inputs. Each nanowire-based XOR gate has structure similar to the XOR tree shown in Fig. 4.

D. Corrector

One-step majority-logic correction is a fast and relatively compact error-correcting technique [15]. There is a limited class of ECCs that are one-step-majority correctable which include type-I two-dimensional EG-LDPC. In this section, we present a brief review of this correcting technique. Then we show the one-step majority-logic corrector for EG-LDPC codes.

1) One-Step Majority-Logic Corrector: One-step majority-logic correction is the procedure that identifies the correct value of a each bit in the codeword directly from the received codeword; this is in contrast to the general message-passing error-correction strategy (e.g., [23]) which may demand multiple iterations of error diagnosis and trial correction. Avoiding iteration makes the correction latency both small and deterministic. This technique can be implemented serially to provide a compact implementation or in parallel to minimize correction latency.

This method consists of two parts: 1) generating a specific set of linear sums of the received vector bits and 2) finding the majority value of the computed linear sums. The majority value indicates the correctness of the code-bit under consideration; if the majority value is 1, the bit is inverted, otherwise it is kept unchanged.

The theory behind the one-step majority corrector and the proof that EG-LDPC codes have this property are available in [15]. Here we overview the structure of such correctors for EG-LDPC codes.

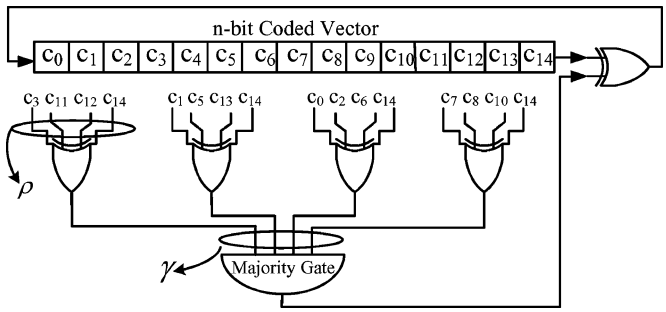

Fig. 7. Serial one-step majority logic corrector structure to correct last bit (bit 14th) of 15-bit (15,7,5) EG-LDPC code.

A linear sum of the received encoded vector bits can be formed by computing the inner product of the received vector and a row of a parity-check matrix. This sum is called *Parity-Check* sum. The core of the one-step majority-logic corrector is generating γ parity-check sums from the appropriate rows of the parity-check matrix. The one-step majority logic error correction is summarized in the following procedure. These steps correct a potential error in one code bit lets say, e.g., c_{n-1}.

1) Generate γ parity-check sums by computing the inner product of the received vector and the appropriate rows of parity-check matrix.
2) The γ check sums are fed into a majority gate. The output of the majority gate corrects the bit c_{n-1} by inverting the value of c_{n-1} if the output of majority gate is "1".

The circuit implementing a serial one-step majority logic corrector for (15, 7, 5) EG-LDPC code is shown in Fig. 7. This circuit generates γ parity-check sums with γ XOR gates and then computes the majority value of the parity-check sums. Since each parity-check sum is computed using a row of the parity-check matrix and the row density of EG-LDPC codes are ρ, each XOR gate that computes the linear sum has ρ inputs. The single XOR gate on the right of Fig. 7 corrects the code bit c_{n-1} using the output of the majority gate. Once the code bit c_{n-1} is corrected the codeword is cyclic shifted and code bit c_{n-2} is placed at c_{n-1} position and will be corrected. The whole codeword can be corrected in n rounds.

If implemented in flat, two-level logic, a majority gate could take exponential area. The two-level majority gate is implemented by computing all the $\binom{\gamma}{\lceil \gamma+1/2 \rceil}$ product terms that have $\lceil \gamma + 1/2 \rceil$ ON inputs and one $\binom{\gamma}{\lceil \gamma+1/2 \rceil}$-input OR-term. For example, the majority of 3 inputs a, b, c is computed with $\binom{3}{2} = 3$ product terms and one 3-input OR-terms as follows:

$$\text{Majority}(a, b, c) = ab + ac + bc. \quad (5)$$

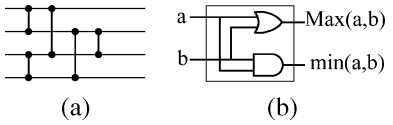

Fig. 8. (a) Four-input sorting network; each vertical line shows a one-input comparator. (b) One comparator structure.

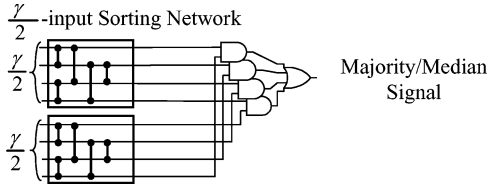

Fig. 9. Eight-input majority gate using sorting network.

In the next section we presents a novel, compact implementation of majority circuits.

2) Majority Circuit Implementation: Here we present a compact implementation for the majority gate using *Sorting Networks* [13]. The majority gate has application in many other error-correcting codes, and this compact implementation can improve many other applications.

A majority function of γ binary digits is simply the median of the digits (where we define the median of an even number of digits as the $\gamma/2+1$st smallest digit).

To find the median of the γ inputs, we do the following:

1) divide the γ inputs into two halves with size $\gamma/2$;
2) sort each of the halves;
3) the median is 1 if for $i=1,2,\ldots,\gamma/2$ the ith element of one half and the $(\gamma/2+1-i)$th element of the other half are both 1.

We use binary *Sorting Networks* [13] to do the sort operation of the second step efficiently. An n-input sorting network is the structure that sorts a set of n bits, using a 2-bit sorter building blocks. Fig. 8(a) shows a 4-input sorting network. Each of the vertical lines represents one comparator which compares two bits and assigns the larger one to the top output and the smaller one to the bottom [see Fig. 8(b)]. The four-input sorting network, has five comparator blocks, where each block consists of two two-input gates; overall the four-input sorting network consists of ten two-input gates in total.

To check the condition in the third step, we use $\gamma/2$ two-input AND gates followed by a $\gamma/2$-input OR gate. Fig. 9 shows the circuit implementing the above technique to find the median value of 8 bits. It has two $\gamma/2$-input (four-input) sorting networks followed by combinational circuitry, consisting of four two-input AND gates and a four-input OR gate, which can be implemented with three two-input OR gates. Therefore in total an eight-input majority gate implemented with sorting networks take 27 two-input gates; in contrast, the two-level implementation of this majority gate takes $\binom{8}{5}=56$ five-input AND gates and one 56-input OR gate.

3) Serial Corrector: As mentioned earlier, the same one-step majority-logic corrector can be used to correct all the n bits of the received codeword of a cyclic code. To correct each code-bit, the received encoded vector is cyclic shifted and fed into to the XOR gates as shown in Fig. 7. The serial majority corrector takes

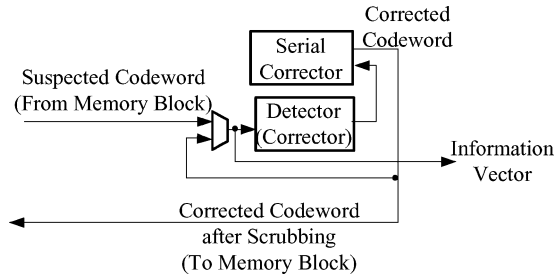

Fig. 10. Partial system overview with serial corrector. Here only the corrector and its detector are shown; these blocks connect to the memory block and the rest of the system shown in Fig. 1.

n cycles to correct an erroneous codeword. If the fault rate is low, the corrector block is used infrequently; since the common case is error-free codewords, the latency of the corrector will not have a severe impact on the average memory read latency. The serial corrector must be placed off the normal memory read path. This is shown in Fig. 10. The memory words retrieved from the memory unit are checked by detector unit. If the detector detects an error, the memory word is sent to the corrector unit to be corrected, which has the latency of the detector plus the n round latency of the corrector.

4) Parallel Corrector: For high error rates [e.g., when tolerating permanent defects in memory words as well (see Section VIII)], the corrector is used more frequently and its latency can impact the system performance. Therefore we can implement a parallel one-step majority corrector which is essentially n copies of the single one-step majority-logic corrector. Fig. 1 shows a system integration using the parallel corrector. All the memory words are pipelined through the parallel corrector. This way the corrected memory words are generated every cycle. The detector in the parallel case monitors the operation of the corrector, if the output of the corrector is erroneous, the detector signals the corrector to repeat the operation. Note that faults detected in a nominally corrected memory word arise solely from faults in the detector and corrector circuitry and not from faults in the memory word. Since detector and corrector circuitry are relatively small compared to the memory system, the failure rate of these units is relatively low; e.g., in a memory system that runs at 1 GHz and $P_f = 10^{-18}$, the parallel corrector for a (255, 175, 17) code should see one upset every 30 hours (calculation is done using the reliability equations in Section VII). Therefore, the error detection and repeat process happens infrequently and does not impact the system throughput.

Assuming our building blocks are two-input gates, γ number of ρ-input parity-check sums will require $\gamma\times(\rho-1)$ two-input XOR gates. The size of the majority gate is defined by the sorting network implementation. Table II shows the overall area of a serial one-step majority-logic corrector in the number of two-input gates for the codes under consideration. The parallel implementation consists of exactly n copies of the serial one-step majority-logic corrector.

Generating the linear binary sums (XORs) of the one-step majority sum is the same as Fig. 4. The majority gate is simply computed following the structure shown in Fig. 9 using the nanowire-based substrate.

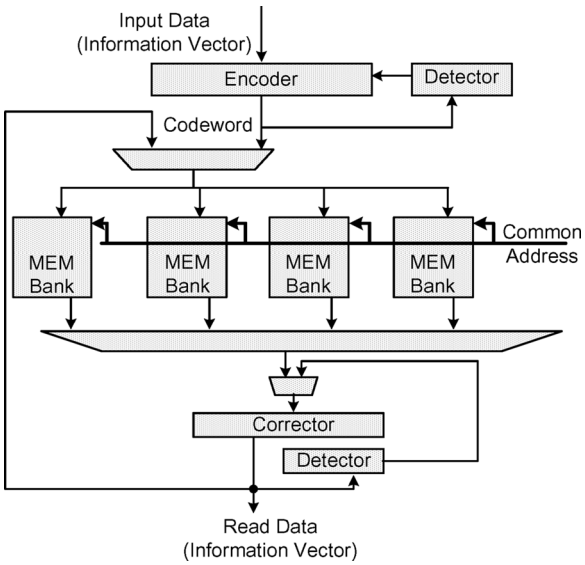

Fig. 11. Banked memory organization, with single global corrector.

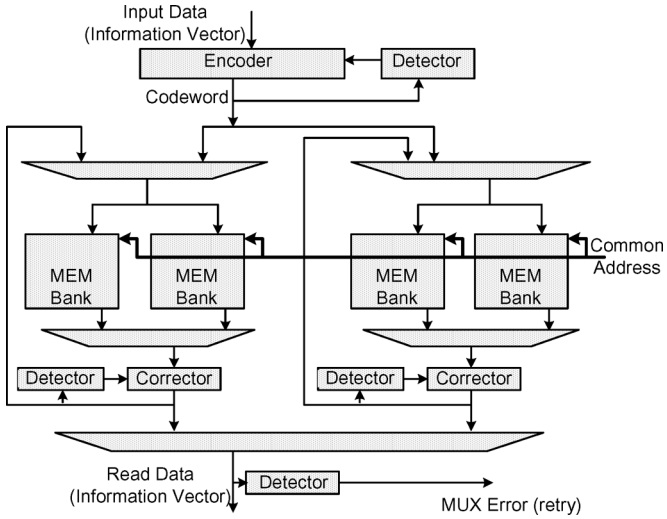

Fig. 12. Banked memory organization with cluster size of 2.

E. Banked Memory

Large memories are conventionally organized as sets of smaller memory blocks called banks. The reason for breaking a large memory into smaller banks is to trade off overall memory density for access speed and reliability. Excessively small bank sizes will incur a large area overhead for memory drivers and receivers. Large memory banks require long rows and columns which results in high capacitance wires that consequently increases the delay. Furthermore long wires are more susceptible to breaks and bridging defects. Therefore excessively large memory banks have high defect rate and low performance. The organization of NanoMemory is not different from the conventional memory organization, except that the overhead per bank is larger due to the scale difference between the size of a memory bit (a single wire crossing) and the support structures (e.g., microscale wires for addressing and bootstrapping). The work presented in [6] provides more detail on memory banks and shows how the banks would be integrated into a complete memory system. The memory system overview shown in Fig. 1 can be generalized to multiple banks as shown in Fig. 11, where similarly the encoder and correctors are protected with fault-secure detectors.

Memory words must be scrubbed frequently to prevent error accumulation. The number of faults that accumulate in the memory is directly related to the scrubbing period. The longer the scrubbing period is, the larger the number of errors that can accumulate in the system. However, scrubbing all memory words serially can take a long time. If the time to serially scrub the memory becomes noticeable compared to the scrubbing period, it can reduce the system performance. To reduce the scrubbing time, we can potentially scrub all the memory banks in parallel. For this, each memory bank requires a separate corrector and detector unit. It may not be necessary to go to the extreme of scrubbing all banks in parallel. Instead, we can cluster a number of memory banks together and consider a corrector and detector unit for each *Cluster*. Fig. 12 shows a memory system with two parallel corrector units. Here each cluster contains two memory banks. In Section IX, we calculate the appropriate cluster sizes to balance performance and area.

VII. Reliability Analysis

In this section, we analyze the reliability of the system. To measure the system reliability, we estimate the probability that system fails—i.e., the system experiences a greater number of errors in a memory word than the number of errors the error-correcting code can tolerate. With this analysis, we then show the impact of protecting the ECC support logic (see Section VII-B).

A. Analysis

We assume the fault probability of each device at each cycle (P_f) has i.i.d. and random distribution over the devices of the memory system. Recall e_e and e_{de} are the nominal number of errors that occurs in encoder and encoder detector during memory write operation at one instance. Similarly, e_m, e_c, and e_{dc} are the number of errors that occur in a memory word and its corresponding corrector and detector. Let n_e, n_c, and n_d be the size of the circuitry involved in an operation on a single code bit in the encoder, corrector, or detector, respectively. This is the size of the logic cone of a single output of each of the above units. For example, in a detector each logic cone is a ρ-input XOR gate generating a single bit of the syndrome vector (see Section VI-B).

Let a nominal unit have a logic cone size of x (x may be one of the values n_e, n_c, n_d). With worst-case analysis the output of the logic cone fails when any of the devices in the logic cone fails. So when at least one of the x devices inside the cone is erroneous, the output of the logic cone, which is a code bit, would be erroneous. Therefore, the probability that a code bit is erroneous in any of the above units is $P_{bit\_circuit} = 1 - (1 - P_f)^x$. Similarly the probability that a memory-bit is erroneous in scrubbing interval of s cycles is

$$P_{bit\_mem} = 1 - (1 - P_f)^{xs} \tag{6}$$

where x is the number of devices contained in one memory cell. When using 6 T-SRAM cells, $x = 6$. When using a NanoMemory, the memory bit is essentially a single nanowire

crosspoint. However, since accessing each memory bit requires reading the signal value through a pair of nanowires (see Fig. 2), the correctness of each memory bit depends on the correctness of two nanowires. Therefore, for the NanoMemory design, $x = 2$. Each unit or a memory-word experience e errors among n bits of the codeword with the probability

$$P_{\text{unit}} = \binom{n}{e} P_{\text{bit}}^{e} (1 - P_{\text{bit}})^{n-e} \tag{7}$$

which is simply a binomial distribution; n is the code-length, P_{bit} is either P_{bit_mem} or $P_{bit\_circuit}$, and e is e_e, e_m, e_c, e_{de}, or e_{dc}.

As explained in Section VI-B, errors in the encoder unit are detected by its following detector, and are corrected by repeating the encoding operation to generate a correct encoded vector. The detector can detect up to $d-1$ errors overall in these two units. Where d is the code distance. With worst-case assumptions, the detector fails to detect the errors if there are more than $d-1$ errors. Therefore, we define the first reliability condition as

$$\text{Cond. I)} \qquad e_e + e_{de} < d$$

which states that the total number of errors in the encoder unit and the following detector unit must be smaller than the minimum distance of the code. The detector of the corrector is also capable of detecting up to $d-1$ errors accumulated from memory unit, corrector unit and the second detector unit. Similarly with worst-case assumption, the detector fails to detect errors when they are more than $d-1$. Therefore, the second reliability condition is defined as

$$\text{Cond. II)} \qquad e_m + e_c + e_{dc} < d$$

Furthermore, the corrector can recover a memory-word with up to $\lfloor (d-1)/2 \rfloor = \lfloor \gamma/2 \rfloor$ errors from the memory unit. If more errors are accumulated in a memory word, then the EG-LDPC codes cannot correct the memory word. Therefore, the third reliability condition is formulated as

$$\text{Cond. III)} \qquad e_m \leq \lfloor (d-1)/2 \rfloor$$

which states that the maximum number of tolerable errors in each memory word is $\gamma/2$. Satisfying the three previous conditions guarantees that the memory system operates with no undetectable or uncorrectable errors. We calculated the probability of each of the above conditions employing (7), for all the various EG-LDPC codes.

Section IX illustrates the reliability of the system for different device failure rate P_f. It also presents the best design points for optimizing reliability, area, and throughput.

B. Impact of Providing Reliability for Supporting Logic

It is important to understand the impact of protecting the supporting logic on the system FIT rate. Could the system FIT rate be low enough if only memory words were protected? What is the potential cost of protecting the supporting logic? We answer these questions with the following example.

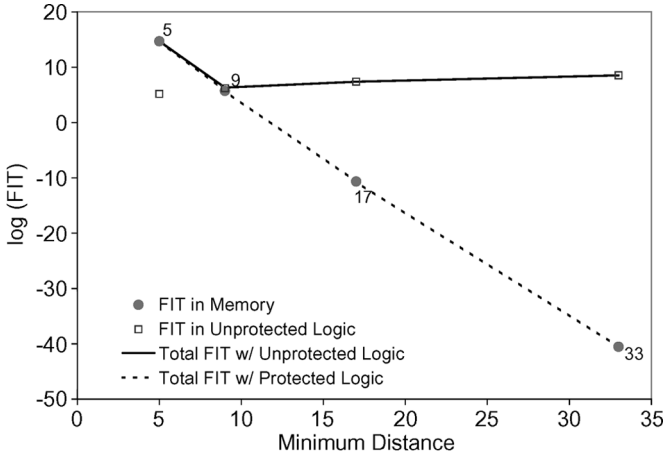

Fig. 13. Impact of protecting logic on system reliability for fault rate equals 10^{-19} per bit per cycle ($P_f = 10^{-19}$).

Fig. 13 shows the FIT rate of the system decomposed into the contribution from the memory bank and the contribution from the supporting logic. The FIT in the supporting logic is without a fault-secure detector (i.e., any error in the supporting logic results an erroneous output, with worst-case analysis). Obviously the FIT of the whole system with no logic protection is the sum of the above two FITs, illustrated with a solid line. This graph is plotted for a device fault rate of $P_f = 10^{-19}$ with a memory scrubbing interval of 10 minutes. As you can see, for codes with minimum distance larger than 9, the FIT of the system with no logic protection is dominated by the FIT of the unprotected logic. Using codes with greater redundancy will decrease the FIT of memory bank; however, since the unprotected logic has a nontrivial FIT rate, increasing the code redundancy without protecting the logic does not decrease the FIT of the composite system. To achieve the higher reliability the logic must also be protected. The FIT of such system with fault secure logic is illustrated with the dashed line, and as you can see, the FIT of this system follows very closely the FIT of the memory bank. Protecting supporting logic, is essentially done by the fault-secure detector and the only cost to achieve the logic protection is the cost that we pay for the detectors. Table III shows that the detector takes a negligible fraction of area compared to the memory core, encoder, and corrector. Therefore with minimal area overhead the system reliability can be greatly improved.

VIII. Tolerating Permanent Defect in Memory Cells

In this paper, so far we focused on a transient fault-tolerant memory structure. Equally important is to tolerate permanent defects in the memory systems. Since the memory unit has a regular structure, most of the defect tolerant designs are based on row and column sparing. This means that we overpopulate the rows and columns based on the expected defect rate so that after removing all the defective rows and columns the memory has the desired size. For example, if we want to have a 1 K $\times$ 1 K memory, and the junction defect rate is 0.001% which results in nanowire defect rate of $P_{\text{wire}} = 1000 \times 10^{-5} = 1\%$, then, with 2% row and column overpopulation (20 more rows

and columns), the system yields a 1 K × 1 K memory core with 99.6% probability

$$\begin{aligned} P_{row\_yield} &= P_{column\_yield} \\ &= \sum_{i=0}^{20} \binom{1020}{i} (P_{\text{wire}})^i (1 - P_{\text{wire}})^{1020-i} \\ &\approx 0.9981 \end{aligned} \tag{8}$$

and the memory yields when both rows and columns yield which is the product of the two probabilities P_{row_yield} and P_{column_yield}

$$P_{row\_yield} \times P_{column\_yield} = 0.9981^2 = 0.996. \tag{9}$$

So when the defect rate is small (0.001%), even with very small area overhead (2% overpopulation), the system can yield a perfect 1 K × 1 K memory core with high probability.

A. ECC for Defect Tolerance

However, with higher defect rates, the column and row sparing can be very costly. For example, a 1% junction defect rate on a nanowire with 1000 junctions implies that, with almost 100% probability, every nanowire in the memory block has at least one defective junction. At this defect rate we cannot afford to discard nanowires with any defective junctions. The work presented in [6] suggests a defect tolerant technique that is more efficient than the column and row sparing. This technique discards nanowires that have more defective junctions than a set threshold. These nanowires will be replaced with spare nanowires. The limited number of defective junctions on the remaining nanowires are tolerated using ECCs.

For the previous example with a junction defect rate of 1%, if we keep nanowires with up to 12 defective junctions, then the system would require only 31% row and column overpopulation to achieve 99% yield. The computation is shown as follows. The probability that a wire is accepted—i.e., has at most 12 defective junction—is

$$P_{\text{wire}} = \sum_{i=0}^{12} \binom{1000}{i} (0.01)^i (0.99)^{1000-i} \approx 0.792. \tag{10}$$

With only 31% column and row sparing we can get 99% yield

$$\begin{aligned} P_{row\_yield} &= P_{column\_yield} \\ &= \sum_{i=0}^{310} \binom{1310}{i} (P_{\text{wire}})^i (1 - P_{\text{wire}})^{1310-i} \\ &\approx 0.9953 \end{aligned} \tag{11}$$

and the final memory yield is

$$P_{row\_yield} \times P_{column\_yield} = 0.9953^2 = 0.9906. \tag{12}$$

In [6], it is suggested that a reliable lithographic-scale encoder and decoder be used to encode and decode memory bits to tolerate limited defective bits in each row. With the techniques introduced here, the encoder and decoder can be built with fault-prone nanowire circuitry as well.

Here we use the same EG-LDPC codes that we use for transient faults for tolerating permanent defects. The error correction capability of the ECC is partitioned between fault tolerance and defect tolerance similar to the approach in [26]. For example, the EG-LDPC code (255, 175, 17) can correct up to eight errors in each memory word. This can be partitioned to tolerate four transient faults and four permanent defect in each code word. With this technique we allow each memory word (code word) to contain up to four defective junctions; with a row width of 1020, this is about 16 defective junctions per row since each row holds four memory words and each memory word tolerates four defects.

This reduces the area overhead compared to solely row and column sparing. The important point is that this area overhead reduction is achieved with almost no extra cost since it uses the structure which already exists for transient fault tolerance—i.e., encoder, corrector, and detector units. The only potential drawback point for this technique is that it increases the effective FIT rate of the memory block relative to the case where we had all 8 errors available to tolerate transient upsets. Therefore to guarantee the desired reliability, codes with larger minimum distance which can tolerate larger numbers of defects and faults must be used. Section IX shows detail results on how the reliability and area overhead costs are balanced.

[25] and [26] present more sophisticated schemes for memory defect tolerance at the cost of reliable, CMOS ECC circuitry and memories to remap or invert blocks. Our scheme minimizes the reliable support needed for the nanoscale memory.

B. Impact on Error Correction Frequency

When we have a limited number of defective junction in memory words, this means that with high probability a memory word has erroneous bits and must be corrected. Therefore, as mentioned in Section III we use a parallel, fully pipelined corrector to prevent throughput loss. The probability that a memory word has a limited number of defects is computed in the following. Remember that the memory words with more than a set threshold are removed. Let the set threshold of the number of defects in each memory word be D_{thr}, and the defect rate be P_d, therefore the memory words have 0 to D_{thr} defective bits. The probability that a memory word is partially defective and requires correction is

$$P_{def\_mem\_word} = \frac{\sum_{i=1}^{D_{\text{thr}}} \binom{n}{i} P_d^i (1 - P_d)^{(n-1)}}{\sum_{i=0}^{D_{\text{thr}}} \binom{n}{i} P_d^i (1 - P_d)^{(n-1)}}. \tag{13}$$

For example for 1% defect rate and tolerating up to four defective bits per memory word, the probability that a 255-bit memory word contains defective bits, and needs correcting, is $P_{def\_mem\_word} = 0.91$. Therefore, 91% of the memory words must be corrected which motivates the use of the parallel and fully pipelined corrector to prevent high throughput loss. In the following section, we show the effect of D_{thr} on area, throughput, and reliability.

IX. Area and Performance Analysis and Results

There are three design aspects that are specifically important when designing fault-tolerant designs: system reliability, area

overhead, and performance. It is important to see how these three factors interact and generate various design points. There are multiple design parameters that determines the balance between reliability, area, and performance. Among these parameters, those that we keep variable in our simulations are as follows:

- maximum number of defects which exist per memory word $(D_{\rm thr})$;
- scrubbing interval (S);
- cluster size of memory banks for scrubbing (C).

For a fixed memory size, bank size, and transient fault rate, we find the right value for $D_{\rm thr}$, S, and C to minimize area while achieving the desired performance and reliability. First, we review the impact of each of these parameters on area, performance, and reliability.

The threshold on the number of defects per memory word effects area, reliability, and performance.

- Increasing $D_{\rm thr}$ reduces the number of spare memory rows (e.g., nanowires) required [6].
- Since $D_{\rm thr}$ defines where we partition the redundancy in the error correcting code between defect tolerance and fault tolerance, larger $D_{\rm thr}$ means that the code has weaker residual capability for tolerating transient faults and therefore has lower system reliability.
- The decrease in reliability which comes from increasing $D_{\rm thr}$ also increases overhead cost for error correction. For fixed ECC, when we reduce the number of bits available for tolerating transient faults, we must scrub more frequently to prevent error accumulation to tolerate the same transient fault rate. However scrubbing more frequently can reduce performance. To reduce the impact on the system performance, we may need to increase the parallelism in the scrubbing operation by increasing the number of corrector unit, which in turn increases the overall area overhead.

The scrubbing interval length S impacts the system performance and the reliability. The longer the scrubbing interval is, the less reliable the system will be, because more errors can accumulate in each memory word during longer scrubbing intervals. Equation (6) shows how scrubbing interval impacts the reliability of each single memory bit. However the shorter the scrubbing interval is, the lower the throughput will be, because the system spends more time performing the scrubbing operation. Below we show how the value of S impacts the system throughput. Assume a memory system has bank size of B and cluster size of C. This means that every C memory banks share one corrector and detector to perform the scrubbing operation. During each scrubbing operation all the $C \times B$ memory words in each cluster will be read, corrected, and written back into the memory. With fully pipelined parallel corrector this takes about $B \times C$ cycles. If the scrubbing interval is S cycles then the system throughput loss is

$$\text{Throughput loss} = \frac{B \times C}{S}. \tag{14}$$

If S is too small, the throughput loss is large.

The impact of memory bank cluster size and memory bank size is also clear on the system throughput: larger memory bank size and cluster size increase the throughput loss. In this paper, we set the memory bank size to 1 K $\times$ 1 K, to achieve high enough memory density following the detail analysis on memory bank size provided in [4] and [6]. Here we vary cluster size to optimize the throughput and area overhead. When the cluster size is large, the parallel corrector is shared among a large number of memory words and therefore the area of the corrector and detector is amortized over a large number of memory words. However, the throughput loss can increase for large cluster size (14).

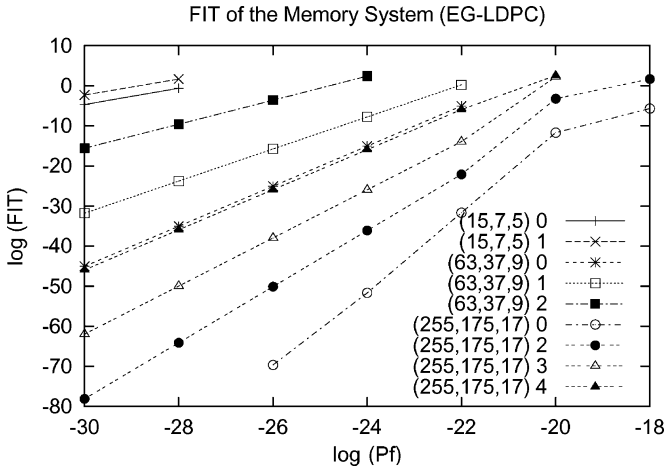

Fig. 14. FIT of EG-LDPC codes for a system with 10^{12} memory bits, memory bank size of 1 Mb, system frequency of 1 GHz, and the defect rate of 1%. The curve labels are of the form: $(n, k, d)D_{\rm thr}$.

For our simulation, we set the limit of throughput loss to $<0.1\%$ and reliability to <1 FIT, and then minimize the area overhead. Fig. 14 shows the reliability of different codes for different $D_{\rm thr}$. We also provided the case with $D_{\rm thr} = 0$, which means that the EG-LDPC codes are solely used for transient faults and the permanent defects are not tolerated with current ECC. Setting the limit on the throughput and reliability, determines the values of S scrubbing interval and C cluster size of memory banks.

Fig. 14 plots the reliability of the systems that satisfy the throughput loss limit and reliability limit while achieving the minimum area overhead. The decomposed area of these design points is shown in Table III. All of the previous design points are for memory size of 10^{12} bits. For these calculations we assume a memory unit with the following parameters: lithographic wire pitch of 105 nm (45 nm node [1]), nanowire pitch of 10 nm, defect rate of 0.01 per memory junction, and memory bank size of 10^6 bits.

Fig. 15 plots the total area per bit for different memory sizes, when the upset rate is 10^{-28} errors/device/cycle. The area of the memory banks are computed following the area model provided in [4]. The area of the supporting units (encoder, corrector, and detector) is computed using the area model of *NanoPLA* provided in [5].

The codes (255, 175, 17) tolerating 4 defects per memory words and (63, 37, 9) tolerating 2 defects, are the most compact designs for EG-LDPC codes. The area overhead of the code in the flat part of the curve is defined by the following multiple factors:

1) code overhead (n/k);

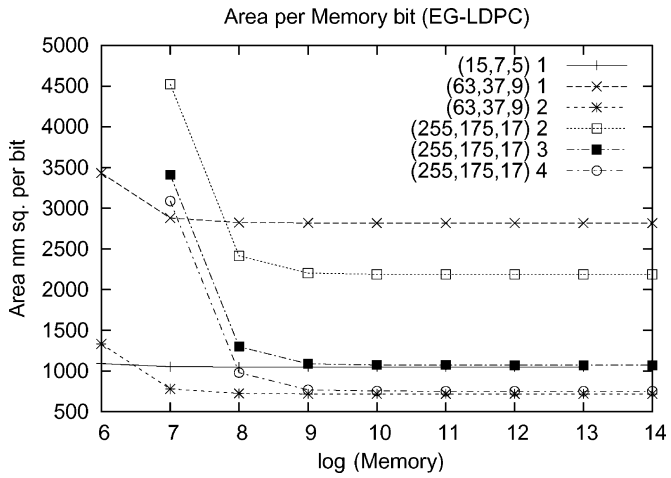

Fig. 15. Area of the memory system versus the memory size; the fault rate is $P_f = 10^{-28}$ errors/device/cycle, the scrubbing interval and the cluster size is set to values that satisfy FIT < 1 and throughput loss $<$0.001. The curve labels are of the format: $(n, k, d)D_{\rm thr}$.

2) constant area overhead of the memory bank which makes the per-bit area of the memory decrease for larger memory size;
3) number of accepted defective junctions per memory word $D_{\rm thr}$.

The final area per bit shown in Fig. 15 is the combined result of the above factors. The code overhead (n/k) is smallest for larger codes; e.g., (255, 175, 17) has the lowest code overhead. If all the other costs where amortized out over the large number of memory bits, we would expect that the largest code shows the lowest area per memory bit. However one important fact that dominates the code overhead is the limited memory bank size. We set the memory banks size fixed at 1 Mb, and for a cluster of memory banks, which can be from 1 to 1000 banks per cluster, we dedicate one corrector and one detector (see Fig. 12). The area overhead of these units are amortized over the memory bits of a cluster and is not reduced as the total memory size increase. The only costs that are reduced as the total memory size grow is the global encoder and its detector unit which are shared among all the memory bits.

Finally, the memory core area per bit plays an important role in the area overhead of the system. This is greatly influenced by $D_{\rm thr}$; the more defective junctions we can tolerate per memory row, the fewer spare rows the system requires and the smaller the area will be. This effect is visible when comparing the same codes [e.g., (255, 175, 17)] for different $D_{\rm thr}$ (e.g., 2, 3, and 4). The combination of all of these factors results in the curves shown in Fig. 15.

In order to understand how each part of the memory system contribute to the final area overhead and how they change with the cluster size, we show decomposed area for a memory system of size 1 Mb, which is essentially one memory bank, in Table IV. First of all, let us look at the memory core area per bit. The memory core area per bit is fixed for the code and $D_{\rm thr}$ pair. It does not change with the cluster size or with the total memory size increase. For one code, the larger number of defects it can tolerate the smaller the area of the core will be, because it can use more defective wires and requires less wire sparing. For example, compare rows 4 and 6 of Table IV. The area of the Final Memory when tolerating 4 defect per codeword is almost 1/3 compared to the case when tolerating 2 defect per codeword (746 versus 2180). The maximum number of defective junctions that the smaller codes can tolerate per row is larger than the maximum number of defective junctions of the larger codes. For our system with memory rows of 1000 bits, a (15,7,5) code which tolerates 1 defect per memory word essentially tolerates 66 errors per row, and a (255,175,17) which tolerates 4 defective bits, tolerates 12 errors per row. Therefore, generally, smaller codes could result in lower area because they can tolerate more defective bits in a row. However larger codes have better code rates (n/k). The combination of these factors makes the memory core using code (63, 37, 9) and tolerating 2 defective junctions, the smallest memory cores; note the second row of Table IV with the per-bit area of 719 nm^2.

TABLE IV
DECOMPOSED AREA PER BIT OF SINGLE CLUSTER (MEMORY SIZE OF 1 Mb). THE UNIT OF AREA IS NANOMETERS SQUARED PER BIT

Code (n,k,d)	Final Memory	Global Enc.+Det.	Cluster Cor.	Cluster Det.	D_{thr}
(15,7,5)	1050	5.41	18.19	3.90	1
(63,37,9)	719	73.8	273	21.4	2
(63,37,9)	2817	73.8	273	21.4	1
(255,175,17)	746	8693	7360	153.0	4
(255,175,17)	1066	8693	7360	153.0	3
(255,175,17)	2180	8693	7360	153.0	2

The second part is the area overhead due to the corrector and detector per bank cluster. The area of the corrector and detector is amortized over the memory bits of one cluster. Table IV shows the area of the corrector and decoder amortized over 1 Mbit-memory bank (one cluster). For larger clusters (e.g., Table III) these net area per memory bit decrease.

The global encoder and its detector, are shared among the whole memory system. Therefore, the net per bit area of these units decrease as the memory size increases. For large enough memory ($\geq$0.1 Gb) these units have negligible area overhead per bit compared to other parts of the system.

The overall area overhead per memory bit for large enough memories, is dominated mainly by the memory core area; the overhead of this core area is determined both by the defect-tolerant overpopulation and rate of the ECC. The cluster corrector and detector also contribute to the area, but their area is a second-order effect compared to the memory core area. The global encoder contributes an even smaller amount to the effective memory bit area. The curves in Fig. 15 plot the total area per bit for a range of memory size, and shows its decrease as the memory size increases.

X. SUMMARY

In this paper, we presented a fully fault-tolerant memory system that is capable of tolerating errors not only in the memory bits but also in the supporting logic including the ECC encoder and corrector. We used Euclidean Geometry codes. We proved that these codes are part of a new subset of ECCs that have FSDs. Using these FSDs we design a fault-tolerant encoder and corrector, where the fault-secure detector monitors

their operation. We also presented a unified approach to tolerate permanent defects and transient faults. This unified approach reduces the area overhead. Without this technique to tolerate errors in the ECC logic, we would required reliable (and consequently lithographic scale) encoders and decoders. Accounting for all the above area overhead factors, all the codes considered here achieve memory density of 20 to 100 Gb/nm^2, for large enough memory ($\geq$0.1 Gb).

ACKNOWLEDGMENT

The authors would like to thank Dr. S. Ghosh for her valuable reference to EG-LDPCs. This material is based upon work supported by the Department of the Navy, Office of Naval Research. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation or the Office of Naval Research.

REFERENCES

[1] ITRS, "International technology roadmap for semiconductors," 2005. [Online]. Available: http://www.itrs.net/Links/2005ITRS/Home2005.htm

[2] Y. Chen, G.-Y. Jung, D. A. A. Ohlberg, X. Li, D. R. Stewart, J. O. Jeppesen, K. A. Nielsen, J. F. Stoddart, and R. S. Williams, "Nanoscale molecular-switch crossbar circuits," *Nanotechnology*, vol. 14, pp. 462–468, 2003.

[3] Y. Chen, D. A. A. Ohlberg, X. Li, D. R. Stewart, R. S. Williams, J. O. Jeppesen, K. A. Nielsen, J. F. Stoddart, D. L. Olynick, and E. Anderson, "Nanoscale molecular-switch devices fabricated by imprint lithography," *Appl. Phys. Lett.*, vol. 82, no. 10, pp. 1610–1612, 2003.

[4] A. DeHon, "Deterministic addressing of nanoscale devices assembled at sublithographic pitches," *IEEE Trans. Nanotechnol.*, vol. 4, no. 6, pp. 681–687, 2005.

[5] A. DeHon, "Nanowire-based programmable architectures," *ACM J. Emerging Technol. Comput. Syst.*, vol. 1, no. 2, pp. 109–162, 2005.

[6] A. DeHon, S. C. Goldstein, P. J. Kuekes, and P. Lincoln, "Non-photolithographic nanoscale memory density prospects," *IEEE Trans. Nanotechnol.*, vol. 4, no. 2, pp. 215–228, Feb. 2005.

[7] A. DeHon and M. J. Wilson, "Nanowire-based sublithographic programmable logic arrays," in *Proc. Int. Symp. Field-Program. Gate Arrays*, Feb. 2004, pp. 123–132.

[8] M. Forshaw, R. Stadler, D. Crawley, and K. Nikolić, "A short review of nanoelectronic architectures," *Nanotechnology*, vol. 15, pp. S220–S223, 2004.

[9] R. G. Gallager, *Low-Density Parity-Check Codes*. Cambridge, MA: MIT Press, 1963.

[10] J. E. Green, J. W. Choi, A. Boukai, Y. Bunimovich, E. Johnston-Halperin, E. DeIonno, Y. Luo, B. A. Sheriff, K. Xu, Y. S. Shin, H.-R. Tseng, J. F. Stoddart, and J. R. Heath, "A 160-kilobit molecular electronic memory patterned at 10^{11} bits per square centimeter," *Nature*, vol. 445, pp. 414–417, Jan. 25, 2007.

[11] S. Hareland, J. Maiz, M. Alavi, K. Mistry, S. Walsta, and C. Dai, "Impact of CMOS process scaling and SOI on the soft error rates of logic processes," in *Proc. Symp. VLSI*, 2001, pp. 73–74.

[12] J. Kim and L. Kish, "Error rate in current-controlled logic processors with shot noise," *Fluctuation Noise Lett.*, vol. 4, no. 1, pp. 83–86, 2004.

[13] D. E. Knuth, *The Art of Computer Programming*, 2nd ed. Reading, MA: Addison Wesley, 2000.

[14] Y. Kou, S. Lin, and M. P. C. Fossorier, "Low-density parity-check codes based on finite geometries: A rediscovery and new results," *IEEE Trans. Inf. Theory*, vol. 47, no. 7, pp. 2711–2736, Jul. 2001.

[15] S. Lin and D. J. Costello, *Error Control Coding*, 2nd ed. Englewood Cliffs, NJ: Prentice-Hall, 2004.

[16] R. J. McEliece, *The Theory of Information and Coding*. Cambridge, U.K.: Cambridge University Press, 2002.

[17] H. Naeimi, "A greedy algorithm for tolerating defective crosspoints in nanoPLA design," M.S. thesis, Dept. Comput. Sci., California Inst. Technol., Pasadena, CA, Mar. 2005.

[18] H. Naeimi, "Reliable integration of terascale designs with nanoscale devices," Ph.D. dissertation, Dept. Comput. Sci., California Inst. Technol., Pasadena, CA, Sep. 2007.

[19] H. Naeimi and A. DeHon, "Fault secure encoder and decoder for memory applications," in *Proc. IEEE Int. Symp. Defect Fault Tolerance VLSI Syst.*, Sep. 2007, pp. 409–417.

[20] H. Naeimi and A. DeHon, "Fault-tolerant nano-memory with fault secure encoder and decoder," presented at the Int. Conf. Nano-Netw., Catania, Sicily, Italy, Sep. 2007.

[21] S. J. Piestrak, A. Dandache, and F. Monteiro, "Designing fault-secure parallel encoders for systematic linear error correcting codes," *IEEE Trans. Reliab.*, vol. 52, no. 4, pp. 492–500, Jul. 2003.

[22] A. Saleh, J. Serrano, and J. Patel, "Reliability of scrubbing recovery-techniques for memory systems," *IEEE Trans. Reliab.*, vol. 39, no. 1, pp. 114–122, Jan. 1990.

[23] M. Sipser and D. Spielman, "Expander codes," *IEEE Trans. Inf. Theory*, vol. 42, no. 6, pp. 1710–1722, Nov. 1996.

[24] D. R. Stewart, D. A. A. Ohlberg, P. A. Beck, Y. Chen, R. S. Williams, J. O. Jeppesen, K. A. Nielsen, and J. F. Stoddart, "Molecule-independent electrical switching in pt/organic monolayer/ti devices," *Nanoletters*, vol. 4, no. 1, pp. 133–136, 2004.

[25] F. Sun, L. Feng, and T. Zhang, "Run-time data-dependent defect tolerance for hybrid CMOS/nanodevice digital memories," *IEEE Trans. Nanotechnol.*, vol. 7, no. 2, pp. 217–222, Mar. 2008.

[26] F. Sun and T. Zhang, "Defect and transient fault-tolerant system design for hybrid CMOS/nanodevice digital memories," *IEEE Trans. Nanotechnol.*, vol. 6, no. 3, pp. 341–351, Jun. 2007.

[27] H. Tang, J. Xu, S. Lin, and K. A. S. Abdel-Ghaffar, "Codes on finite geometries," *IEEE Trans. Inf. Theory*, vol. 51, no. 2, pp. 572–596, Feb. 2005.

Helia Naeimi (S'00–M'08) received the Ph.D. and M.Sc. degrees in computer science from the California Institute of Technology, Pasadena, and the B.Sc. degree in computer engineering from Sharif University of Technology in 2008, 2005, and 2002 respectively.

Since 2008, she has been a Research Scientist with Intel Research, Santa Clara Laboratory, Santa Clara, CA. Her main research interest is designing reliable systems with nanotechnology devices, which includes solving challenging issues such as high soft-error rate, low system yield, and high parameter variations.

André DeHon (S'92–M'96) received the S.B., S.M., and Ph.D. degrees in electrical engineering and computer science from the Massachusetts Institute of Technology, Cambridge, in 1990, 1993, and 1996, respectively.

Since 2006, he has been an Associate Professor with the Department of Electrical and Systems Engineering, University of Pennsylvania, Philadelphia. From 1996 to 1999, he co-ran the BRASS Group in the Computer Science Department, University of California at Berkeley, Berkeley. From 1999 to 2006, he was an Assistant Professor with the Department of Computer Science, California Institute of Technology, Pasadena. He is broadly interested in how we physically implement computations from substrates, including VLSI and molecular electronics, up through architecture, CAD, and programming models. He places special emphasis on spatial programmable architectures (e.g., FPGAs) and interconnect design and optimization.