Fault Tolerant Nano-Memory with Fault Secure Encoder and Decoder

Helia Naeimi
Computer Science
California Institute of Technology
Mail Stop: 256-80
Pasadena, CA 91125 helia@caltech.edu

Andre DeHon
Electrical and System Engineering
University of Pennsylvania
200 S. 33rd Street
Philadelphia, PA 19104 andre@acm.org

ABSTRACT

We introduce a nanowire-based, sublithographic memory architecture tolerant to transient faults. Both the storage elements and the supporting ECC encoder and corrector are implemented in dense, but potentially unreliable, nanowire-based technology. This compactness is made possible by a recently introduced *Fault-Secure* detector design [18]. Using Euclidean Geometry error-correcting codes (ECC), we identify particular codes which correct up to 8 errors in data words, achieving a FIT rate at or below one for the entire memory system for bit and nanowire transient failure rates as high as 10^{-17} upsets/device/cycle with a total area below 1.7× the area of the unprotected memory for memories as small as 0.1 Gbit. We explore scrubbing designs and show the overhead for serial error correction and periodic data scrubbing can be below 0.02% for fault rates as high as 10^{-20} upsets/device/cycle. We also present a design to unify the error-correction coding and circuitry used for permanent defect and transient fault tolerance.

1. INTRODUCTION

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 [14, 11] 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, not just the storage and communication systems. Therefore, to build fault-tolerant nanoscale systems, we must protect both combinational logic and memory against transient faults. In the present work 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 and decoder (corrector) circuitry).

Our proposed system with high fault-tolerant capability is feasible when the following two fundamental properties are satisfied: 1) Any single error in the encoder or corrector circuitry can only corrupt a single codeword digit (*i.e.*, cannot propagate to multiple codeword digits). 2) There is a *Fault Secure* detector (FSD) circuit which can detect any limited combination of errors in the received codeword or the detector circuit itself. Property 1 is guaranteed by not sharing logic between the circuitry which produces each bit. The FSD (Property 2) is possible with a more constrained definition for the ECC (Section 2).

Figure 1 shows the memory architecture based on this FSD. There are two FSD units monitoring the output vector of the encoder and corrector circuitry. If an error is detected at the output of the encoder or corrector units, that unit has to redo the operation to generate the correct output vector. Using this detect-and-repeat technique we can correct potential transient errors in the encoder or corrector output to provide a fault-tolerant memory system with fault-tolerant supporting circuitry.

The fault-tolerant capability of this design is as follows: 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, memory-word, and corrector. With fault secure detector we guarantee that the system can correct any error combination as long as $e_m \leq E$, $e_e + e_{de} \leq D$, and $e_m + e_c + e_{dc} \leq D$, where e_{de} and e_{dc} are the number of errors in two separate detectors monitoring the encoder and corrector units. In contrast, the fault-tolerant capability in the existing state-of-the-art design is only guaranteed for $e_m \leq E$ and $e_e = e_c = 0$. The conventional strategy only works as long as we can expect the encoding, decoding, and checking logic to be fault-free, which would prevent the use of nanoscale devices.

It is important to note that transient errors accumulate in the memory words over time. In order to avoid error accumulation which exceeds the code correction capability, the system must *scrub* memory frequently to remove errors. Memory scrubbing is periodically reading memory words from the memory, correcting any potential errors, and writing the corrected words back into the memory (*e.g.*, [19]). The frequency of scrubbing must be determined carefully. The scrubbing frequency impacts the throughput from two directions: 1) The memory cannot be used on scrubbing cycles, reducing the memory bandwidth available to the application; more frequent scrubbing increases this throughput loss effect. 2) During the normal operation, when an error is detected in a memory word, the system must spend a number of cycles correcting the error; these cycles also take

Nano-Net 2007 September 24-26, 2007, Catania, Italy.
 ISBN 978-963-9799-10-3
DOI 10.4108/ICST.NANONET2007.2029

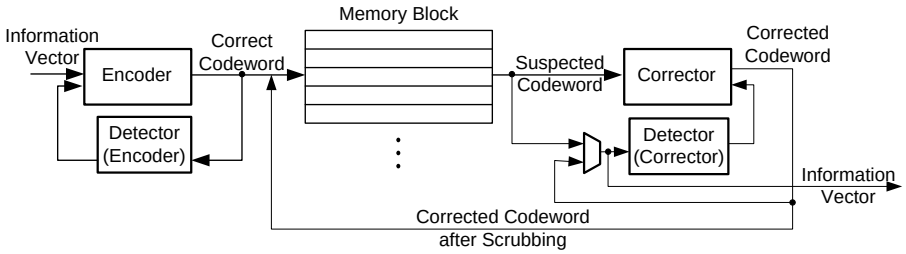

Figure 1: This figure shows the overview of our proposed fault-tolerant memory architecture.

bandwidth away from the application. When scrubbing happens less frequently, more errors accumulate in the memory, and therefore more memory reads require error correction, increasing bandwidth loss.

In this article we show how the FSD corrector introduced in [18] can be implemented in fault-prone nanowire-based logic and used to protect the nanowire-based memory introduced in [5] against transient upsets in both the memory and the logic (Section 4). We further show how to determine the optimum combination of bank size, corrector parallelism, and scrubbing interval for this memory system (Section 5).

Before presenting this design and analysis, we review the fault-secure detector design in Section 2 including a brief review of ECC. Section 3 then reviews the nanowire-based memory architecture.

2. ECC AND FSD REVIEW

2.1 Error-Correcting Codes

Since we build upon linear ECCs, this section briefly reviews the terminology and composition of linear block ECCs. Let $i = (i_0, i_1, ..., i_{k-1})$ be k-bit information vector that will be encoded into n-bit codeword, $c = (c_0, c_1, ..., c_{n-1})$. For linear codes the encoding operation essentially performs the following vector-matrix multiplication: $c = i \times G$, where G is a $k \times n$ generator matrix. Checking the validity of a received encoded vector is done by employing 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: $s = c \times H^T$. The $(n-k)$-bit vector s is called *syndrome* vector. A syndrome vector is zero if c is a valid codeword and non-zero 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 [17]. With this definition, the generator matrix of a systematic code must have the following structure: $G = [I : X]$, 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 decoder circuitry 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 *Cyclic code* if for any codeword c, all the cyclic shifts of the codeword are still valid codewords.

2.2 Fault-Secure Detector

In this work we use a fault secure detector capable ECC (FSD-ECC) to design a fault-tolerant memory system. The FSD-ECC is an error-correcting code with a more constrained definition that is introduced in [18]. The FSD-ECC definition is as follows: 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 $e \leq d-1$, has syndrome vector of weight at least $d-e$.*

Theorem I is proved in [18].

The standard ECC definition simply requires that the syndrome have a non-zero weight. The stronger requirement here guarantees that a limited number of detector errors will still result in a non-zero syndrome so the erroneous codeword is detected.

The above theorem is valid when any single error in the detector circuitry can corrupt at most one output (one syndrome bit). This can be easily satisfied by guaranteeing that the logic circuit of the syndrome bits do not share any nodes; therefore, any single error in the circuit corrupts at most one syndrome bit.

2.3 Euclidean Geometry Codes

Type-I two dimensional *Euclidean Geometry* (EG) codes [16] are interesting for several reasons: (1) they are FSD-ECC as shown in [18], (2) they are *One-Step Majority* correctable, meaning they admit to simple and low latency correction, and (3) they are *cyclic codes* meaning we can implement compact serial correctors. Here, we briefly review these codes and properties. The parity-check matrix for EG codes is an $n \times n$ matrix, and each row of H has weight ρ, and each column has weight γ. This guarantees that the complexity of the parity-check calculation is linear in the product $\gamma \times n$, and that the fanin to any single syndrome bit is limited to γ, resulting in low latency for the computation. The rows of H are not necessarily linearly independent. The rank of H is $(n-k)$ which makes the code of this matrix an (n,k) linear code. Lin and Costello [16] show an algorithm for the construction of these codes and demonstrate that H is a Low-Density Parity-Check matrix, and therefore the code is an LDPC code [12]; henceforth we call these codes, EG-LDPC codes. All the parameters of the EG-LDPC codes are summarized in Table 1. Type-I two-dimensional EG-LDPCs are *One-Step Majority* correctable up to $\gamma/2$ errors, which means that a codeword can be corrected one code bit at a time up to $\gamma/2$ errors, with γ ρ-input XOR gates followed by a single γ-input majority gate (see Section 4). This corrector can be arbitrarily parallelized, up to the width of the codeword, using multiple copies of the single-code-bit correction logic.

Table 1: Parameters of EG-LDPC for any $s \geq 2$.

Information bits (k)	$2^{2s} - 3^s$
Length (n)	$2^{2s} - 1$
Minimum distance (d)	$2^s + 1$
Dimensions of the H	$n \times n$
Row weight of the H (ρ)	2^s
Column weight of the H (γ)	2^s

3. BACKGROUND

3.1 Nano-Memory and NanoPLA

Nanoelectronic molecular technologies have made it possible to design a very compact memory array, and small memory arrays have been successfully fabricated using these techniques [1, 13]. A memory architecture based on a nanowire substrate is developed in [5] that can achieve greater than 10^{11} b/cm^2 density even after including the lithographic-scale address wires. This design uses a nanowire crossbar to store memory bits and a limited number of lithographic scale wires for address and control lines. Figure 2(a) shows 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 [3]. The nanowires can be uniquely addressed from lithographic wires using a programmable deterministic address decoder, shown in the top and right side of the picture. However to program the deterministic decoder, the nanowires passing through each junction must be uniquely selected as well; this is achieved by using a stochastic decoder [20, 9]. The stochastic decoder is used only for bootstrapping and configuring the deterministic decoder. Once the deterministic decoder is programmed, only this decoder is used for normal operation. The area model accounting all the lithographic-scale addressing wires is provided in [5]. Here we use that area model, and compute the area overhead of the fault tolerant design following that model. To implement the supporting logic we use the *nanoPLA* architecture model [6]. It has a regular structure like conventional PLA and implements logic in consecutive NOR planes.

4. FAULT-TOLERANT NANO-MEMORY

In this section we provide the detail design for the proposed fault-tolerant memory system (shown in Figure 1) implemented on a sub-lithographic nanowire-based substrate. This section starts by showing the detail of each unit (detector, corrector, and encoder) and then shows how these units can be integrated together in nanowire-based substrates and connected to the memory unit.

Large memory systems are partitioned into a collection of limited-size banks to avoid long wiring in large memories which reduce both yield and performance. If we further give each memory bank a separate corrector unit, they can be scrubbed in parallel, reducing scrubbing latency. However, since scrubbing does not require the encoder unit, all the memory banks share a single encoder unit. This is illustrated in Figure 2(b).

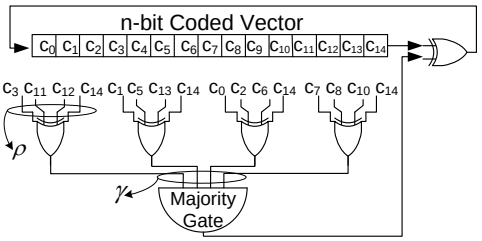

Figure 3: One-step majority corrector for $(15, 7)$ code.

4.1 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: $s = c \times H^T$, and therefore each bit of the syndrome vector is the product of the following vector-vector multiply: $s_i = c \cdot {h_i}^T$, where ${h_i}^T$ is the transposed of the ith row of the parity-check matrix. The above product is a linear binary sum over digits of c where the corresponding digit in h_i is 1. This binary sum is implemented with an XOR gate. 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 in a tree structure. For the whole detector, it takes $n(\rho - 1)$ 2-input XOR gates.

An error is detected if any of the syndrome bits has a non-zero 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. In order to avoid a single point of failure, we must implement the OR gate in a reliable substrate—*i.e.*, we use a lithographic-scale wired-OR. This n-input wired-OR is much smaller than implementing the entire $n(\rho - 1)$ 2-input XORs at the lithographic scale. The area of each detector is computed using the model form [6] and [5] accounting all the supporting lithographic wires. Details of this implementation are provided in the Appendix.

4.2 One-Step Majority Corrector

For EG-LDPC, the one-step majority-logic corrector corrects up to $\gamma/2$ error bits in the received encoded vector. It computes γ ρ-bit *parity-check* sums, which are simply implemented with a ρ-input XOR function each, similar to the ρ-input XOR gates of the detector. The majority value of the parity-check sums are then evaluated, with a γ-input majority gate. Figure 3 shows a corrector for the $(15, 7)$ EG-LDPC code. If the majority value is 1 then the code bit under consideration holds an erroneous value and has to be inverted. When majority value is 0 the code bit is correct. Since EG-LDPC are cyclic codes, a single majority corrector circuit can be used for all the code bits [16]. To correct each code bit, the received encoded vector is cyclically shifted and fed into the XOR gates. If implemented in flat, two-level logic, a majority gate could take exponential area. Naeimi and DeHon [18] shows a compact implementation of a majority gate using *Sorting Networks* [15].

4.3 Encoder

The encoder structure of a systematic code calculates the parity function of each parity bit based on the information bits. Each parity function is an XOR gate similar to the detector. Therefore the encoder circuitry consists of $(n - k)$

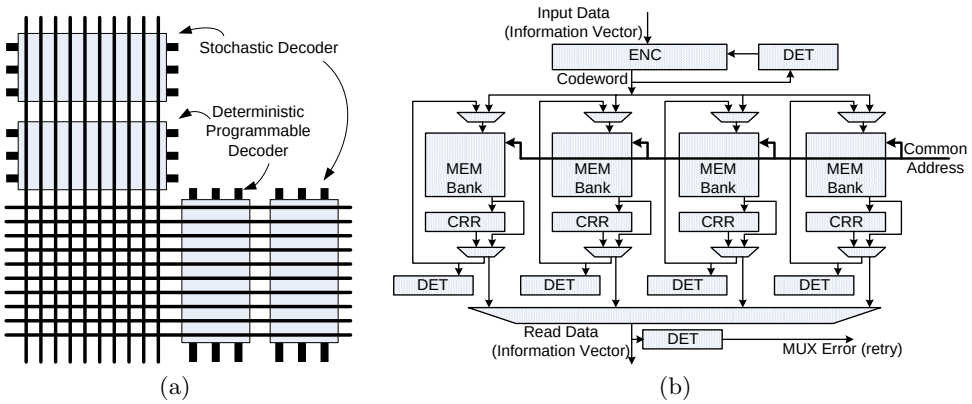

Figure 2: (a) Nano-Memory, (b) Memory Banking

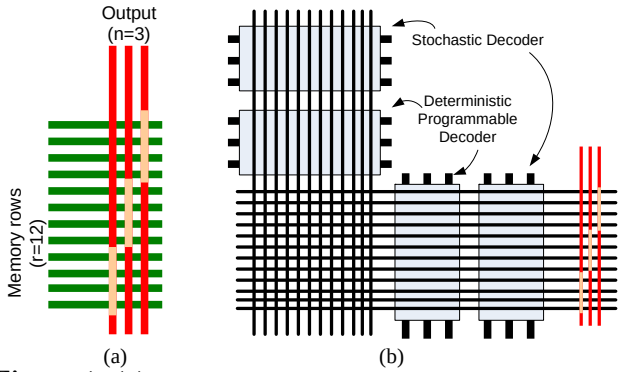

Figure 4: (a) Shows a simple DEMUX for $n = 3$, and $r = 12$. (b) Shows integration of this DEMUX with nano-memory.

XOR gates. The exact fan-in size of the XOR gates is determined in [18]. The area of the encoder is further computed using the area model in [5] and will be analyzed further in Section 5.

4.4 Memory Access with NanoScale Interface

The nano-memory architecture introduced in [5] has a lithographic scale interface. For our fault-tolerant system the supporting logic is implemented at the sub-lithographic scale; we replace the lithographic scale interface with a sub-lithographic one. The main part of the interface is a demultiplexer designed with gate-able nanowires [2], shown in Figure 4(a). Each of the output nanowires (vertical wires) is gate-able by r/n nanowires of the memory rows, where r is the number of memory rows. For example, in Figure 4(a), the four topmost rows gate the first output nanowire. The deterministic lithographic wires selecting the memory rows are programmed to select exactly one of the r/n controlling nanowires of each output nanowire. This way at most one nanowire is driven high and can actually gate the controllable region. With this design, on each read, the demultiplexer selects n rows and presents them to the output nanowires; these outputs are then routed to the encoder, corrector and detectors. Figure 4(b) shows the integrated demultiplexer with the nano-memory system. With a few additional transformations, the demultiplexer can be statistically assembled similar to the restoration array used by the nanoPLA [7, 4].

5. SYSTEM ANALYSIS

5.1 Performance Analysis

As noted in the introduction, high scrubbing rates will decrease the time spent correcting data at the cost of additional cycles lost to scrubbing, while low scrubbing rates do the opposite. In this section, we quantify these effects to determine the balance which yields the optimum scrubbing interval. We further explore the appropriate level of banking and corrector parallelism to contain throughput loss. Finally, we note how this design can accommodate both permanent memory defects and transient memory upsets.

As you can see in Figure 1, when there is no error in the memory word, the memory words are pipelined through the detector and therefore we can read one word per cycle without any throughput loss. If we select our memory word size and scrubbing rate appropriately, the vast majority of read operations can take this fast path. However, when the detector observers an error, the memory word must pass through the corrector to be corrected. The total number of cycles that will be lost is equal to the latency of the corrector and the detector (the dataflow of the detector must be flushed); for our serial corrector design, the main latency is due to the corrector. For larger codes, where the serialized corrector can take a long time, multiple copies of the corrector can be used to reduce the throughput loss.

Scrubbing frequency also impacts the rate at which correction is required; longer scrubbing intervals increase the number of errors accumulated in the memory and therefore more retrieved memory words have to go through the correction operation. Figure 5 shows the throughput loss due to correction for various parallelism in the correction. This system is of size 1 Gbit and frequency of 10 GHz; it is protected by a $(63, 37)$ EG-LDPC code and the device failure rate is 10^{-20} faults/device/cycle.

The throughput lost to scrubbing cycles has the opposite effect as the scrubbing interval decreases. When the system scrubs more frequently, it has to spend more time performing the scrubbing operation. When there are multiple memory banks in the system where each has its own corrector, the scrubbing operation happens in parallel, and the throughput loss is divided by the number of memory banks. Figure 5, shows the throughput loss due to scrubbing with separate curves for various *logical* bank sizes. The logical bank size is the size of the memory that share one corrector.

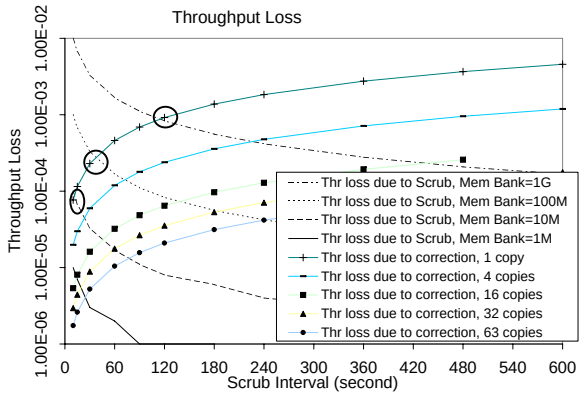

Figure 5: Throughput Loss for 1 Gbit memory system operating at 10 GHz using a (63,37) code at $P_f = 10^{-20}$.

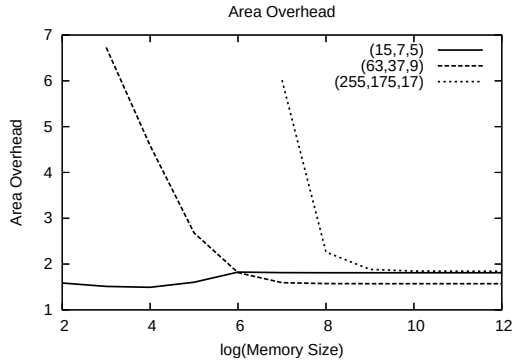

Figure 6: Area overhead for different codes.

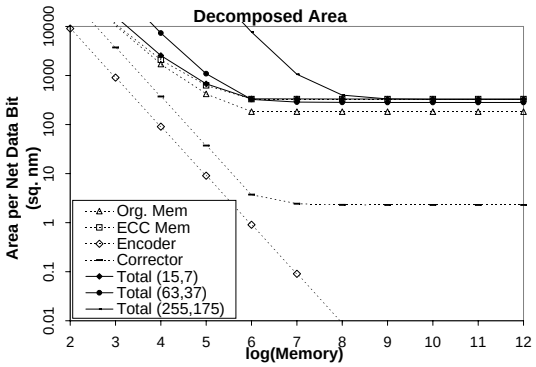

Figure 7: Decomposed area for $(15, 7)$ code and total area for all the three codes.

Since the practical *physical* bank size is around 1 Mbit [8], the logical bank size of *e.g.*, 10 Mbit, essentially consists of 10 physical bank of size 1 Mbit sharing a detector and corrector. The optimum scrubbing interval is circled for each logical bank size for the fully serialized corrector case. With multiple parallelism and banking curves (Figure 5), there are multiple configurations which achieve the same throughput loss (*e.g.*, the configurations with correction parallelism and memory banks size pair of (16 copy, 100 Mbit) and (1 copy, 10 Mbit) both have a throughput loss around 10^{-4}). For a given throughput loss target, we can then select the configuration which requires least area. In this example, the (16 copy, 100 Mbit) configuration is smaller.

5.2 Area and Reliability Analysis

In this section we analyze the area overhead of the fault-tolerant memory system using the banking structure explained in Section 4 and illustrated in Figure 2(b). There are three parts contributing to the area overhead of the fault-tolerant memory: 1) The code overhead (n/k), 2) The encoder and its detector, 3) The corrector and its detector. Figure 7 shows the area/bit of each of the above parts vs. the memory size for the (15,7) code. In this model, the maximum memory bank size is limited to 1 Mbit. For system with memory size smaller than 1 Mbit, the whole memory is one bank. Since the area of the encoder is amortized over a larger number of memory bits, the encoder area/bit decreases as memory size increases. Since each memory bank has a separate corrector and detector, the corrector area/bit is impacted by memory bank size. The area/bit of the corrector remains flat after the memory size exceeds the fixed memory bank size of 1 Mbit, as does the memory area/bit. The area/bit decrease of the original memory and the fault-tolerant memory as the bank size increases is due to the fact that the lithographic scale wires, mainly used for address lines, will be amortized over a larger number of memory bits at the larger bank sizes. In this area analysis the full pitch for nanowires is 10 nm, the full pitch of the lithographic-scale wires is 108 nm, and the substrate is assumed defect free. In practice, permanent nanowire defects would force us to overpopulate both the rows and the columns [8]; to first-order, this impacts the population of both the raw memory and the corrected memory by similar factors.

The area overhead, which is the area of the fault-tolerant memory plus the area of the encoder, corrector and detectors divided by the area of the original memory is plotted in Figure 6. If the system were implemented with one bank and there were no limitation on the bank size, we would expect that, at large memory sizes, the dominant area overhead factor would be due to the code overhead (n/k), and therefore the smallest to largest area overhead were associated to $(255, 175)$, $(63, 37)$, and $(15, 7)$ codes respectively similar to the data presented in [18]. However the limited bank size prevents us from reaching the point where code rate overhead completely dominates these other overheads. With limited bank size the area of the lithographic-scale wires are not completely amortized over the memory bits, and therefore the ratio of the ECC memory to the original memory is less than the code overhead.

The other factor that determines the area overhead in the flat part of the curve is the area overhead of the corrector and its detector compared to each memory bank size. This is more visible for the larger codes (*e.g.*, $(255, 175)$) since the area of the corrector and detector is not negligible compared to the area of the memory bank. The above factors determines the area overhead factor illustrated in Figure 6 which is different than the code overhead. Here, the $(63, 37)$ code provides the most compact implementation.

The reliability of such a system is developed in [18]. We computed the reliability analysis of the system using the analysis from [18] with the encoder, corrector, detector and memory cell size of the current system. The reliability in *FIT* (Failure In Time (10^9 hours)) is plotted in Figure 8 for various codes and fault rates.

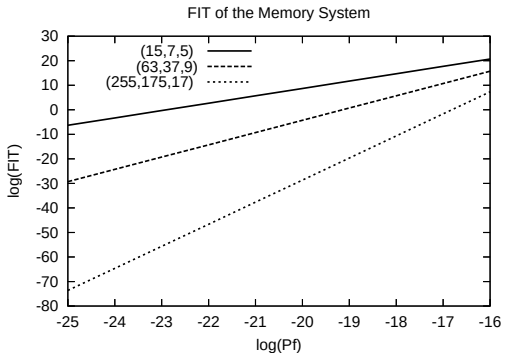

Figure 8: FIT of the memory system.

5.3 Permanent Defect Tolerance

DeHon et al. [8] suggest a strategy to tolerate defective crosspoints in the memory crossbar: when a nanowire has more than a set threshold number of defective junctions the entire nanowire will be discarded and replaced with a spare. Consequently to achieve a fixed memory size, the nanowire crossbar has to be overpopulated with nanowires, so that the remaining crossbar after removing the defective wires has the desired size. If the crosspoint defect rate is small enough, then removing nanowires that have any defective crosspoint does not required large redundancy overhead for overpopulating; *i.e.*, only a small number of nanowires will be removed. For example, with our 1K×1K memory banks, a crosspoint defect rate of 0.001% would add approximately 1% to the nanowire defect rate. However, when the defect rate is high, removing all the nanowires with defective junctions can result in an unreasonably large area overhead. A 0.05% crosspoint defect rate on a nanowire with 1000 junctions implies that each nanowire has roughly a 50% chance of including a defective junction. Therefore for high crosspoint defect rates, we only discard nanowires with more defective junctions than a set threshold, and we use ECC to tolerate the limited number of defective junctions on the remaining nanowires [8].

Here we use the same EG-LDPC code to tolerate errors due to the defective junctions and transient faults. This forces us to partition the error correction capabilities of the code between permanent defects and transient upsets. For instance, we might allocate 4 of the 8 tolerable memory word errors available in the (255,175,17) code to permanent defects and 4 to transients. This means we discard nanowires containing any memory words with more than 4 permanent defects; we end up with a transient fault capability in the memory of only 4 memory bits, increasing the effective FIT rate of the memory block relative to the case where we had all 8 errors available to tolerate transient upsets.

For these high defect rate scenarios, many words will contain at least one defect and require correction. This makes correction a common event meaning correction throughput is now important. Consequently, we employ a fully parallel and pipelined corrector to correct each memory word as it is read from memory. Since the corrector is pipelined there will be no extra latency accessing the corrector. This fully parallel corrector is implemented on the nanotechnology substrate for compact design, and consequently it is susceptible to transient faults. Therefore a fault secure detector must monitor the operation of this corrector block. When the detector identifies an error, it is due to a transient fault in the corrector or detector circuitry and repeating the correcting operation will fix the error. Note that this repeating operation will take latency and cause some throughput loss, but since it is only due to the errors in the corrector circuit, it happens with low frequency and does not cause high throughput loss. To economize area, only one corrector is fully parallelized and this one is used to correct all the memory words read from the memory core in normal operation. During the scrubbing operation, each bank has its own serialized corrector.

Full characterization of the unified design that tolerates these permanent defects and faults across a range of defect rates is beyond the scope of this paper and will be developed in future work.

6. SUMMARY

In this paper we have presented a fault-tolerant nanotechnology memory system that tolerates faults in the encoder, corrector and detector circuitry as well as the memory. We use Euclidean Geometry codes with a fault-secure detector to design this memory system. We demonstrate particular codes which tolerate up to 8 errors in the stored data and up to 16 total errors in memory and correction logic with an area less than 1.7 times the unprotected memory area; we determine an optimum scrubbing interval, banking scheme, and corrector parallelism so that error correction has negligible performance overhead. We further note that this design shows how we may be able to use a nanoscale corrector to tolerate permanent crosspoint defects.

Acknowledgments

This research was funded in part by National Science Foundation Grant CCF-0403674 and the Defense Advanced Research Projects Agency under ONR contract N00014-01-0651. 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.

7. REFERENCES

[1] 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*, 14:462–468, 2003.

[2] Y. Cui, X. Duan, J. Hu, and C. M. Lieber. Doping and Electrical Transport in Silicon Nanowires. *Journal of Physical Chemistry B*, 104(22):5213–5216, June 8 2000.

[3] Y. Cui, L. J. Lauhon, M. S. Gudiksen, J. Wang, and C. M. Lieber. Diameter-Controlled Synthesis of Single Crystal Silicon Nanowires. *Applied Physics Letters*, 78(15):2214–2216, 2001.

[4] A. DeHon. Law of Large Numbers System Design. In S. K. Shukla and R. I. Bahar, editors, *Nano, Quantum and Molecular Computing: Implications to High Level Design and Validation*, chapter 7, pages 213–241. Kluwer Academic Publishers, Boston, 2004.

[5] A. DeHon. Deterministic Addressing of Nanoscale Devices Assembled at Sublithographic Pitches. *IEEE Transactions on Nanotechnology*, 4(6):681–687, 2005.

[6] A. DeHon. Design of Programmable Interconnect for Sublithographic Programmable Logic Arrays. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, pages 127–137, February 2005.

[7] A. DeHon. Nanowire-Based Programmable Architectures. *ACM Journal on Emerging Technologies in Computing Systems*, 1(2):109–162, 2005.

[8] A. DeHon, S. C. Goldstein, P. J. Kuekes, and P. Lincoln. Non-Photolithographic Nanoscale Memory Density Prospects. *IEEE Transactions on Nanotechnology*, 4(2):215–228, 2005.

[9] A. DeHon, P. Lincoln, and J. Savage. Stochastic Assembly of Sublithographic Nanoscale Interfaces. *IEEE Transactions on Nanotechnology*, 2(3):165–174, 2003.

[10] A. DeHon and M. J. Wilson. Nanowire-Based Sublithographic Programmable Logic Arrays. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, pages 123–132, February 2004.

[11] M. Forshaw, R. Stadler, D. Crawley, and K. Nikolić. A Short Review of Nanoelectronic Architectures. *Nanotechnology*, 15:S220–S223, 2004.

[12] R. G. Gallager. *Low-Density Parity-Check Codes*. MIT Press, 1963.

[13] J. E. Green et al. A 160-Kilobit Molecular Electronic Memory Patterned At 10^{11} Bits per Square Centimetre. *Nature*, 445:414–417, January 25 2006.

[14] J. Kim and L. Kish. Error Rate In Current-Controlled Logic Processors With Shot Noise. *Fluctuation and Noise Letters*, 4(1):83–86, 2004.

[15] D. E. Knuth. *The Art of Computer Programming*. Addison Wesley, second edition, 2000.

[16] S. Lin and D. J. Costello. *Error Control Coding*. Prentice Hall, second edition, 2004.

[17] R. J. McEliece. *The Theory of Information and Coding*. Cambridge University Press, 2002.

[18] H. Naeimi and A. DeHon. Fault Secure Encoder and Decoder for Memory Applications. In *Proceedings of the IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, September 2007.

[19] A. Saleh, J. Serrano, and J. Patel. Reliability of Scrubbing Recovery-Techniques for Memory Systems. *IEEE Transaction on Reliability*, 39(1):114–122, 1996.

[20] S. Williams and P. Kuekes. Demultiplexer for a Molecular Wire Crossbar Network. United States Patent Number: 6,256,767, July 3 2001.

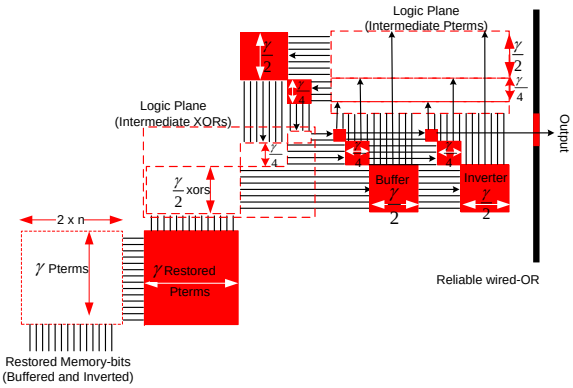

Figure 9: A γ-input XOR tree implemented on nanoPLA structure.

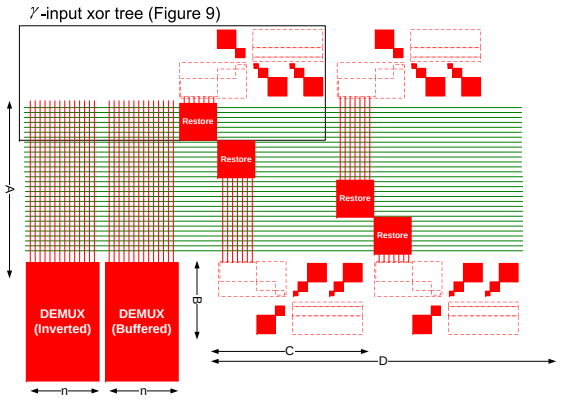

Figure 10: A detector circuit implemented on nanoPLA: The parameters in the figure are $A = n \times \gamma$, $B = 2 \times Pl(\gamma - 1)$, $C = Pl(2 \times \gamma - 2) + 2 \times Pl(\gamma - 1)$, and $D = n/2 \times C$.

APPENDIX

Figure 10 shows the implementation of the detector on a nanoPLA substrate. The framed block shows a γ-input XOR gate, implementing a $\log_2(\rho)$-level XOR tree in spiral form (Figure 9). The solid boxes display the restoration planes and the white boxes display the wired-OR planes of nanoPLA architecture model [10, 6]. The signals rotate counter clockwise, 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 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.