

# Combining Static and Dynamic Defect-Tolerance Techniques for Nanoscale Memory Systems

Susmit Biswas, Gang Wang, Tzvetan S. Metodiev\*, Ryan Kastner, Frederic T. Chong  
University of California, Santa Barbara, CA, 93117, USA

University of California, Davis, CA, 95616, USA\*

{susmit,chong}@cs.ucsb.edu, {wanggang,kastner}@ece.ucsb.edu, tsmethodiev@ucdavis.edu

**Abstract**— Nanoscale technology promises dramatically increased device density, but also decreased reliability. With bit error rates projected to be as high as 10%, designing a usable nanoscale memory system poses a significant challenge. In particular, we need to bootstrap a sea of unreliable bits into contiguous address ranges which are preferably as large as 4K-byte virtual memory pages. We accomplish this bootstrapping through a combination of dynamic error correction codes within 32-bit blocks and a static defect map which tracks usability of these blocks. The key insight is that statically-determined defect locations can be much more powerful than dynamically correcting for unknown locations, but that defect maps are only practical at a coarse granularity. Using a combination of BCH error correction codes and a Bloom-Filter-based defect map, we achieve a memory efficiency of 60% and 13% for 4K-byte pages at 1% and 10% bit-error rates, respectively.

## I. INTRODUCTION

Prototype nanoscale devices have been constructed around the world using both lithography and self-assembly technology. It is evident that in the not too distant future, we will be able to create computing devices with density reaching  $10^{11}$  devices per square centimeter [2]. Nanoscale devices, however, are expected to have high defect rates. These defects can be roughly divided into two classes: (i) permanent defects caused by inherent physical uncertainties in the manufacturing process, and (ii) transient faults due to lower noise tolerance or charge injection at reduced voltage and current levels. While the exact manufacturing defect rate is not yet pinpointed, defect rate up to 10% has been reported[11]. This is eight orders of magnitude worse than the one per billion defect rate found in current CMOS technology. With a bit-error rate as high as 10%, building reliable, large nanoscale memories of billions of bits poses a significant challenge.

A number of defect-tolerant design methods have been proposed [4][16][19] to deal with high error rates. These methods either use redundancy such as Error Correcting Codes (ECC) [10], or reconfiguration in post manufacturing process to map out the defective regions. Error correction provides better reliability, but at very high error rates the overhead also becomes very large. Therefore, reconfiguration has been reported as the most reliable option [3] for high defect rates, but these are targeted towards reconfigurable devices where discovering the defective components is required only before logic reconfiguration. For a nanoscale memory system, the overhead of keeping a reconfiguration bit for every non-

reliable memory bit negates the density advantage offered by nanoscale memory devices as the overhead becomes greater than 100% due to larger size of reliable reconfiguration bit. Therefore, a more compact way of storing the reconfiguration data is needed. Wang et al. [19] proposed the use of a Bloom filter [1] for storing the defect map. As the bit error rate increases, however, the Bloom filter will need a large amount of reliable memory to store the bit-level defect map. To create a memory system compatible with commodity systems, our goal is to design a memory architecture that can provide reliable, contiguous blocks of storage to support virtual memory pages. Without dynamic error correction, the probability of finding a usable 4K-byte memory page with 10% bit error rate is  $3.7781 \times 10^{-188}$ , which motivates us to use error correcting codes in memory.

In this paper, we use an analytical model for computing the defect rate of memory. We use error correcting codes to provide block-level correctness of memory, and addresses of any bad blocks are stored in a Bloom filter based static defect map. The higher granularity of the information stored in a Bloom filter decreases the amount of reliable memory required for the Bloom filter by a factor of  $n$  in the case of  $n$ -bit wide block. We also develop a new hashing scheme for the Bloom filter which reduces the false positive rate and computational overhead. Furthermore, we analyze the use of spare blocks to enhance the system reliability producing 60% and 13% memory efficiency at 1% and 10% bit error rate respectively compared to near zero yield in conventional memory system with 4K-byte size pages.

The rest of the paper is organized as follows. In section II, we give a description of the related research work. In Section III, we present an overview of our system model and assumptions. We discuss the efficacy of a range of error correcting codes in Section IV and Bloom filters in Section V. Our methodology and results are presented in Sections VI and VII respectively. Finally, we conclude with a perspective of our future work.

## II. RELATED WORK

In this section, we describe the research in the field of dynamic error correcting codes, static reconfiguration, and hybrid approaches for developing reliable memory.

Error correcting schemes have been widely used for both memory architectures and communication beginning with Von Neumann’s seminal work on repetition codes [12]. Jeffery et al.[8] proposed a 3 level error correcting memory architecture for nanoscale memory using a single or double error correcting codes, 4<sup>th</sup> level RAID and sparing of RAID arrays. Ou et al. [13] proposes efficient hardware design for Hamming encoder and decoder for enhancing memory reliability at the cost of only 5*n.s* delay in the memory access time. Hamming codes, however, are capable of correcting a single error in the block of physical bits used in the encoding, and they become less productive for high error rates. For high error rates, stronger and multiple error correcting codes such as BCH codes are required as investigated by Sun et al. for nano-scale devices [16]. While error correction does not introduce significant overhead for low bit error rates, strong error correcting codes such as the BCH(250,32,45) are required to overcome bit error rates as high as 10% as expected in future nano-scale memory devices. The reliance, however, on such codes introduces a large overhead in both area and latency and negates the density and processing advantages offered by future nano-scale devices.

To reduce the overhead, we can combine active error correction through encoding with defect maps to store the locations of faulty bits in memory [18]. For reconfigurable architectures, tile based memory units have been proposed where components store the defect map in a distributed fashion [3][5][21].The drawback of using defect maps is that the storage overhead is usually very high. Tahoori [17] proposes a *defect unaware design flow* which identifies universal defect free subsets within the partially defective chips and reduces the size of the required defect map. Wang et al. [19] proposes the use of Bloom filters for storing defect maps for nanoscale devices. Hashing for every bit, however, is expensive computationally and may significantly increase the memory access times. The authors in [16] propose the use of employing CMOS memory for storing metadata to identify good parts of the memory using two schemes: (i) a two level hierarchy of CMOS and nano-device memory; (ii) a *bootstrapping* technique to store the reliable block information in some good part of the non-reliable memory and storing this index in the reliable CMOS. The amount of memory to store the ranges increases with the sparseness of faulty memory bits. Moreover, to build a memory compliant with commodity virtual memory systems, we need to have fixed-size pages instead of variable-size ranges. In this paper, we propose a combination of static defect map and dynamic error correction technique in order to provide large, reliable, contiguous regions of addressable memory.

### III. MODEL

In a memory with manufacturing defects, it is possible that entire regions with many bits become defective during device manufacturing. However, the exact pattern and flux of defects is currently unknown. Therefore, we assume that

defects will be distributed in a random fashion which is a much worse situation than tackling correlated errors as the amount of metadata to store defect information becomes large. We also make the simplification that the interconnects between each nanobit and the computational logic are not defective. Kuekes et al. [9] addresses the issue of interconnect reliability by using error correction techniques in interconnect designing for CMOS-nanoscale device hybrid memory.

Our technique is not targeted to a particular nanoscale technology. Rather, we address the reliability issues expected of many of the proposed nanoscale technologies, such as Molecular Memories, Polymer Memories, nanowire crossbar and many others as described in [6]. We assume that testing for defects is feasible in memory, which is beyond the scope of this paper. Mishra et al. [11] propose a scalable *two phase* probabilistic testing technique using tiling test pattern. Chen He et al. [5] propose a two-phase group testing technique for memory-based nanofabrics. Once the defects are identified, our system is able to produce satisfactory reliability from the non-reliable memory through the system design techniques which we evaluate in this paper.

### IV. ERROR CORRECTION CODES

A memory access is performed on a block of bits instead of a single one, and the size of a block depends on the system. In a 32-bit system, the yield of 32-bit block is only 3.43% when the bit defect probability in the memory approaches 10%. In this paper, We will refer *yield* as the usable fraction of the memory. Figure 1(a) shows the yield as a percentage of good blocks given different block lengths and increasing bit defect probability. It can be inferred that we need to use error correction in order to design a usable memory architecture using unreliable nanoscale devices, . In the following subsections, we review some popular error correcting codes and then compare them with respect to computational resource requirement as well as the cost of their implementation.

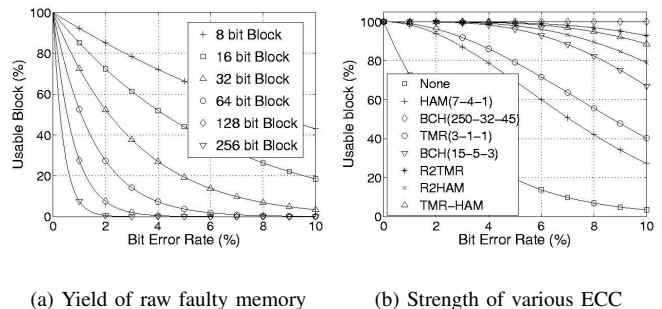


Fig. 1. Yield of defective nanoscale memory decreases with block length and defect probability. Strength of error correcting codes are compared for selecting appropriate code corresponding to a target defect probability.

#### A. Common Error Correcting Codes

Error correcting codes encode the state of a set of bits into a number of physical bits conforming to some specific rules of construction so that errors can be detected and corrected at

decode time. In the following subsections, we denote an error correcting code by  $[n, k, t]$  where it encodes  $k$  data bits into  $n$  physical bits and can correct up to  $t$  errors. *Information rate* [10] of the code is defined by the fraction  $k/n$ .

1) *Single Error Correcting Codes*: Triple modular redundancy (TMR) encodes the state of a single bit over three faulty memory units and a majority voting gate determines the correct bit value. If one of the three faulty components fails the other two systems can correct and mask the fault. On the other hand, if the voter fails then the complete system will fail unless voting circuit is replicated too.

Hamming codes [10] can correct single-bit errors, and can detect up to double-bit errors.

2) *Reed-Solomon Codes*: Reed-Solomon codes (RS) [10] rely on the fact that any  $k$  distinct points uniquely determine a polynomial of degree at most  $k-1$ . If the locations of the faulty symbols are not known in advance, a Reed Solomon code can correct up to  $\frac{n-k}{2}$  erroneous symbols. RS Codes perform well when the code length is large. Traditionally, RS codes are used for bulk media storage because the encoding technique makes them attractive for errors that appear in bursts. Our design, however, is based on the assumption that errors in nanoscale devices are random in nature, thus we neglect burst error correcting codes like Reed-Solomon Codes from our analysis.

3) *Bose-Chaudhuri-Hocquenghem Code(BCH)*: BCH (Bose, Ray-Chaudhuri, Hocquenghem) code are widely studied error correcting codes [10]. In technical terms, a BCH code is a multilevel, cyclic, error-correcting, variable-length digital code used to correct multiple random error patterns.

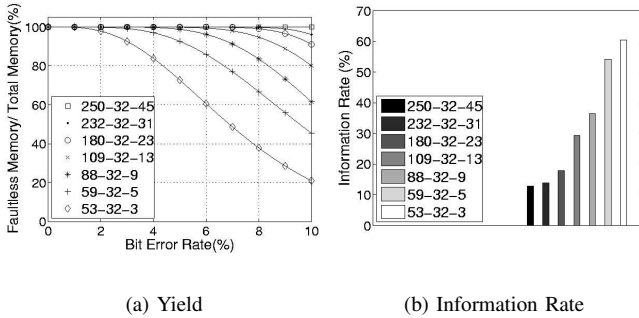


Fig. 2. Comparison of yield and information rate of BCH codes for 32-bit block of data. Stronger BCH code tolerates higher bit defect probability at the cost of lower information rate

BCH codes provide the advantage of generating arbitrary length codes where a  $[n, k, t]$  BCH code can be converted to a  $[n-s, k-s, t]$  code choosing  $s$  such that  $k-s$  is equal to 32. We show the error correction performance for different BCH code sizes in Figure 2(a). To make the decoding process faster, BCH decoding can be parallelized [15].

4) *Concatenation of Error Correcting Codes*: Large error correcting codes such as the BCH  $[250, 32, 45]$  code that allow correction of multiple errors are computationally costly to implement within a feasible memory architecture. To reduce the cost of encoding and decoding, we can concatenate simple

codes, where a number of encoded bits, can be recursively encoded again using a different error correcting code. Consider a code  $C_1$  which encodes  $k_1$  bits to  $m_1$  bits and another code  $C_2$  which encodes  $m_1$  bits to  $m_2$  bits. Therefore, the information rate of the concatenated code becomes  $\frac{k_1}{m_2}$ . We see in Figure 1(b) that the performance of recursive TMR (R2TMR) using 2 levels of recursion is comparable to the performance of the BCH code, where the R2TMR code can be implemented with less overhead.

## B. Comparison of Error Correction Codes

The probability  $P_{block}$  of having a fully working block of length  $L_b$  when the bit error rate is  $p_b$  using an  $[n, k, t]$  code is given by:

$$P_{block} = \left( \sum_{i=0}^t C_i^n p_b^i (1-p_b)^{n-i} \right)^{L_b/k}, \quad (1)$$

where  $C_i^n$  is number of possible selections of  $i$  items from a set of  $n$  items. The behavior of Equation 1 for the above mentioned error correcting codes is plotted in Figure 1(b). We see that the BCH  $[250, 32, 45]$  code provides very good error correction capability compared to other codes. The other codes which are comparable to it are recursive triple modular redundant code using two levels of recursion and the Hamming  $[7, 4, 1]$  code on top of a single level of TMR. The BCH  $[250, 32, 45]$  code seems to be the best candidate for our target bit error rate of 10%. Hence, for further analysis, we use the results of the BCH  $[250, 32, 45]$  code, but even simpler concatenated codes can be used with low overhead for moderate defect probabilities.

## V. BLOOM FILTERS FOR MEMBERSHIP QUERIES

A Bloom filter [1] is a hash-based data structure that offers a compact way to store a set of items, and supports membership queries.

### A. Bloom Filters Background and Practice

A Bloom filter works as follows: A set of  $n$  elements  $S = \{s_1, s_2, \dots, s_n\}$  is mapped to the Bloom filter vector  $B$  of  $m$  bits by a set of  $k$  independent hash functions,  $\{H_1, H_2, \dots, H_k\}$ . Each item in the set  $S$  is hashed  $k$  times, with each hash yielding a bit location in the Bloom filter string that is set to 1. To check if element  $x$  belongs to the set, we can hash it  $k$  times and check if all of the corresponding bits in the Bloom filter are set to 1, otherwise  $x$  is not in the set.

The space efficiency of Bloom filters comes with some percentage of false-positives since  $x$  may hash to bits in the Bloom filter that have been set by a different element. Therefore, Bloom filters are good data structures when membership queries are needed from a stored list of items and the effect of some false-positives can be mitigated. The false-positive rate is a function of the length of the Bloom filter string as it relates to the number of the hashed keys ( $n$ ), the number of hash functions ( $k$ ), and the hash functions' ability to evenly

populate the Bloom filter. The sparser the Bloom filter string, the better the false-positive rate. Given perfect hashing the false positive rate  $FP$  can be approximated by:

$$FP = \left(1 - e^{-\frac{kn}{m}}\right)^k, \quad (2)$$

where  $m$  is the length of the Bloom filter string. Equation 2 is minimized at  $k = \ln(2) \times \frac{m}{n}$ , however, in practice perfect hashing is not possible especially one that is also inexpensive in hardware. This is particularly important when the hardware is used to implement a cache or main memory where limiting read and write latencies is integral.

Bloom filter hash functions are of type *universal hash functions* [7] where each hash function aims to minimize the probability that two input strings will map to the same output string. The practical performance of this hashing scheme has been investigated in Reference [14]. Given a  $b$ -bit key  $X = \langle x_1, x_2, \dots, x_b \rangle$  as input that represents the address of a faulty block, the  $j$ 'th hash function over  $X$  is defined as follows:

$$H_j(X) = (R_1^j \bullet x_1) \oplus (R_2^j \bullet x_2) \oplus \dots \oplus (R_b^j \bullet x_b), \quad (3)$$

where  $R_i^j$  is a random coefficient ranging between 1 and  $m$ . The problem with this approach is that each hash function indicates which one of the  $m$  bits in the Bloom must be set to 1, thus the false-positive rate is improved as the number of hash functions is closer to the optimal number. The storage requirements for  $k$  hash functions is therefore  $(k \times b \times \log_2(m))$  bits. There also seems to be a significant number of correlations between the hashing results from one key to the next due to the fact that the same random numbers  $R_i^j$  from each hash function are used for every key. This limits the possible locations in the Bloom filter string that can be set to 1 making the effective Bloom filter string smaller and denser, thus causing higher false positive rate. To reduce the false positive rate to approximately 3%, the Bloom filter string must be at least a factor of 10 greater than the number of keys hashed in the Bloom filter.

### B. New Hashing Technique

In this paper, we have modified the Bloom filter hashing implementation to use a single hash function that requires maximum memory of  $(b \times \log_2(m))$  bits, and with each hashed key, at most  $b$  bits instead of one are set in the Bloom filter. Given  $b$  random numbers  $\{R_1, R_2, \dots, R_b\}$  each in the range 1 to  $m$ , our single hash function is defined as follows:

$$H(X) = (\epsilon_{R_1} \bullet x_1) \oplus (\epsilon_{R_2} \bullet x_2) \oplus \dots \oplus (\epsilon_{R_b} \bullet x_b), \quad (4)$$

where  $R_i$  is also a number between 1 and  $m$  such that  $R_i' = X \oplus R_i$ , and  $\epsilon_{R_i'}$  is an  $m$ -bit string with only the  $R_i'$ 'th entry set to 1. The  $\oplus$  operation between each key and each random number  $R_i$  ensures that each  $R_i$  is relatively unique for each key, thus allowing us to have at most  $b$  bits in the Bloom filter string set to 1 at random for each key. The false-positive rate in our new hashing approach can be approximated to be

$$FP_{new} = (1 - e^{-nb/m}),$$

which approaches zero as  $b$  approaches zero if  $m$  and  $n$  are kept constant. In practice, however, the false-positive rate is optimal when  $b$  is exactly the number of bits needed to represent each key being hashed. This can be seen from the definition of the hash function in Equation 4, where unnecessary key overlap is introduced if  $b$  is smaller than the length of the binary string for each key and the higher order bits of two keys are the same. Our experimental results

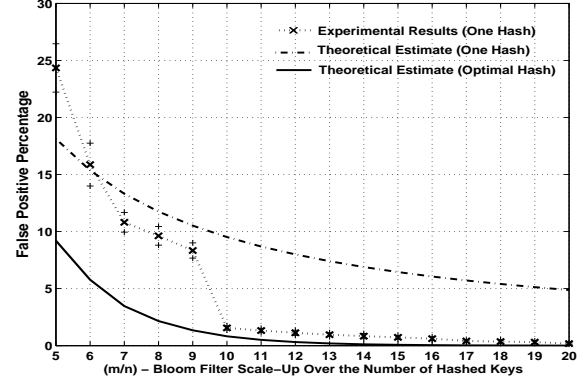


Fig. 3. Experimental Bloom filter results with our new hashing method compared to the existing theoretical estimates. The false-positive rate has a sudden drop at  $m/n = 10$  since the number of bits in the Bloom filter string is exactly equal to the number of memory blocks.

are compared to the traditional theoretical results in Figure 3, where the  $x$ -axis shows the scale-up of the Bloom filter defined as  $m/n$  and the  $y$ -axis shows the false-positive percentage. After  $m/n = 10$ , the Bloom filter string is sparse enough such that we begin to approach the theoretical lower bound for an optimal number of hash functions with our single hash function. At 32 bits per memory block,  $m/n = 10$  gives us a Bloom filter string equal to 3% of the total memory if the failure rate per 32-bit block is 10% and approximately  $1.56\% \pm 0.18\%$  false-positives. As long as the scale factor  $m/n$  is less than the number of bits required for the address of each faulty block (i.e.  $b$  bits), the Bloom filter will offer an advantage over storing all faulty addresses in a table directly. At  $m/n = 20$  the false-positive rate drops to approximately  $0.18\% \pm 0.06\%$  percent and still take less than 6.5% of the total Memory. Perhaps most importantly, using this simple hash function, we can implement a fast Bloom filter suitable for memory architectures.

## VI. A HYBRID APPROACH

While the use of Bloom filters for identifying faulty nanoscale bits [19] is promising, the scheme at the bit-level does not produce high yield in memory as the probability of finding a set of contiguous bits becomes pretty low with increasing bit error rate. More importantly, we concentrate on improving the reliability of memory blocks that will allow memory systems to continue to operate at the level of pages which can be as large as 4 KB each. To keep error correction overhead as small as possible, we choose to concentrate on improving the reliability of 32-bit memory blocks. In a 32-bit system, all the memory accesses are aligned to a 32-bit

boundary. We use sufficiently strong error correcting codes as described in Section IV to increase the reliability of blocks which consist of 32-bits such that the yield of usable pages in memory can be as high as possible.

The basic idea is to use those pages of memory which are completely good and to never use pages which have bad blocks in it. To have higher rate of faultless pages without storing any mapping of faulty blocks to good blocks, we allocate 1 spare block for every  $n$  blocks. If more than one block is bad in that region, the page becomes unusable. If the block error rate is  $r$ , the page size is  $S$  blocks, and we keep 1 spare block for every  $n$  blocks, then the probability of a page being usable is  $P(n, S) = \left( \sum_{i=0}^1 C_i^n r^i (1-r)^{n-i} \right)^{S/n}$ . This probability increases for smaller page sizes. Hence, the performance of the paging system increases if we allow variable page sizes as proposed by Witchel et al. in Reference [20]. We formulate the probability of obtaining a correct page and present the result for different page sizes in the Section VII. For lower error rates we use simpler error correcting codes which provide higher information rate. As the hash computation can be performed in parallel with the decoding process, the overhead of using Bloom filter to determine if a block is bad or not is not perceptible. Moreover, hashing can be pipelined and hence hash results can be obtained during every cycle. In Figure 4, we show a high-level overview of our proposed memory system. Bloom filter based hash function module answers the set-membership query on the input memory address and determines if the memory location is faulty. The error correcting module chooses a block according to the result of the Bloom filter. Optionally, a buffer or cache can be maintained in the memory module to improve the system performance.

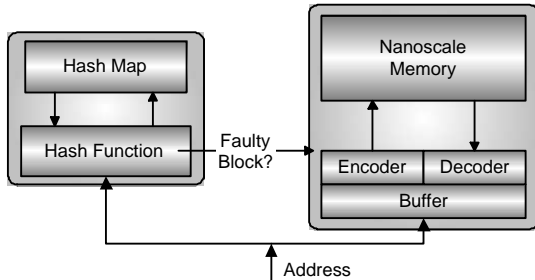


Fig. 4. Block diagram of the memory System. ECC module enhances the block level reliability and Bloom filter stores the defect map. When a memory location is addressed, the Bloom filter is queried in parallel with accessing the memory to hide hashing latency in it.

## VII. RESULTS

### A. Evaluation of Our Approach

Using the fast Bloom filter scheme described in Section V, we can represent random defects in a memory with  $2^{30}$  bits having 10% bit defect rate using as little as 3% of the total memory. This compares to 60% of the total memory needed by bit-level Bloom filters as described in Reference [19]. Raising the granularity of Bloom filters, however, means that the lower-level bits must be protected dynamically through error correction, and this comes with its own additional cost.

Using BCH [250, 32, 45] error correcting code, the fault rate

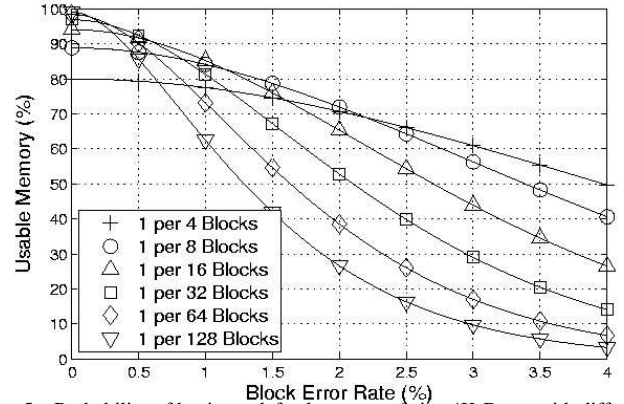


Fig. 5. Probability of having a defectless page of size 4K-Bytes with different number of spare blocks. Having more spare blocks increases the probability, but the overhead also increases.

of 32-bit wide blocks is reduced to 0.0044% and for pagesize of 4K-Bytes, we show that 98% of the pages are usable when the block error rate is 10%. Unfortunately strong error correcting codes such as BCH [250, 32, 45] codes cannot provide high information rate, e.g. the information rate of the BCH [250, 32, 45] code is only 0.1280. For lower bit failure rates we can use codes with higher information rate. In

TABLE I

ECC USED FOR DIFFERENT ERROR RANGES

Defect Rate(%)	Error Correcting Code	Information Rate
0.00 - 1.00	BCH(53,32,3)	0.6038
1.00 - 3.00	BCH(59,32,5)	0.5424
3.00 - 4.00	BCH(88,32,9)	0.3636
4.00 - 6.00	BCH(109,32,13)	0.2936
6.00 - 7.00	BCH(180,32,23)	0.1778
7.00 - 8.00	BCH(232,32,31)	0.1379
8.00 - 10.00	BCH(250,32,45)	0.1280

Table I, we summarize the codes used for different error rate region and their information rates. Figure 5 implies that to get satisfactory fraction of the memory to be usable, we need the block error rate as low as possible. In all the above cases, the error correcting code was chosen such that the block error rate is smaller than 1%. This way we ensure that we get significant number of faultless memory pages (> 77%). We choose to use 1 block per 64 blocks which results in  $\approx 90\%$  yield of pages. In Figure 6 the overall memory yield has been shown. We observe that using only Bloom filter to store the bad block information provides correctness, but the fraction of faultless pages are very close to *zero* for 4K-Bytes pages for error rates of much less than 1% whereas our system shows graceful degradation in performance. Using spare block, reliability is further enhanced as the chance of not having a single bad block increases exponentially with bit error rate.

### B. Overhead

There are two costs associated in the scheme: overhead due to Bloom filter hashing and Error Correcting code. We have to pay high computational overhead of using complex error

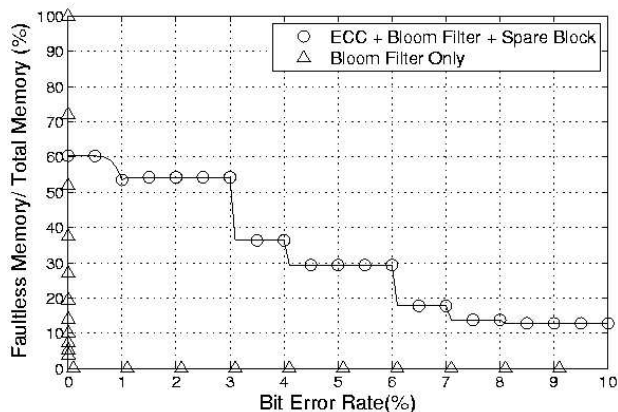


Fig. 6. Percent of defectless memory with 4K-Bytes pages. The yield obtained in our technique beats the yield in conventional memory by several orders of magnitude.

correcting code. Longer BCH codes show very good error correcting strength, but the computation takes much longer time. For bit error rates less than 4%, simpler error correction codes such as concatenated TMR and Hamming(7,4,1) code or recursive Hamming(7,4,1) code with 2 levels of recursion could be used which have low overhead compared to BCH codes. Bloom filter manages the information of bad blocks, but hashing is computationally expensive. To alleviate this overhead, we have described a low cost Bloom Filter implementation with a single hashing function that allows us to compute the hash values with low storage and delay.

### VIII. CONCLUSION

In our work we establish the requirement of using strong error correcting codes in the presence of bit error rates as high as 10%, and analyze the use of other error correcting schemes for lower bit error rates. We evaluate the use of Bloom filter-based defect map storage above the error correction hardware embedded in memory and use spare blocks to increase the amount of usable pages in memory. We find that using just a single spare block for every 64 blocks nullifies the need for having separate map for bad addresses. Through the combination of these techniques, it is possible to build functioning memory using highly unreliable nanoscale technology without sacrificing the density advantages offered by such devices. In particular, we find that we can use relatively inexpensive concatenation of low-overhead error correcting codes for bit error rates as high as 4%. For higher bit error rates we use more complex error correction, but achieve approximately 13% usable memory at the page level for bit error rates as high as 10%. It is expected that CMOS technology will suffer high defect rates when the feature size approaches 20 nm, however, novel technologies such as Carbon Nanotubes and Phase-Change Memory promise feature size of 1-5 nm. At 13% usable memory at the page level, this translates to an area reduction of 2 - 50 times over existing memory devices.

### IX. FUTURE WORK

In our system, memory access incurs added delay of error correction which makes this system unsuitable fast memory

components such as cache. We plan to inspect the scope of decreasing latency by parallel implementation or pipelining technique. For storing the defect map in Bloom filter, we use small amount of reliable memory, but that can be further reduced by chaining of unreliable blocks and by means of bootstrapping from a smaller reliable memory. Our work is based on the assumption that errors will have random distribution throughout the memory. Extending this work, we aim to inspect better techniques for handling correlated errors in memory.

### REFERENCES

- [1] Burton H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM*, 13(7):422–426, 1970.
- [2] A. DeHon, S. C. Goldstein, P. Kuekes, and P. Lincoln. Nonphotolithographic Nanoscale Memory Density Prospects. *IEEE Transactions on Nanotechnology*, 4:215–228, March 2005.
- [3] A. DeHon and K. K. Likharev. Hybrid CMOS/Nanoelectronic Digital Circuits: Devices, Architectures, and Design Automation. In *ICCAD '05: Proceedings of the 2005 IEEE/ACM International Conference on Computer-Aided Design*, pages 375–382, 2005.
- [4] J. Han, J. Gao, Y. Qi, P. Jonker, and J. A. B. Fortes. Toward Hardware-Redundant, Fault-Tolerant Logic for Nanoelectronics. *IEEE Design & Test*, 22(4):328–339, 2005.
- [5] C. He, M. Jacome, and G. de Veciana. Scalable Defect Mapping and Configuration of Memory-Based Nanofabrics. In *IEEE International High Level Design, Validation and Test Workshop (HLDVT)*, 2005.
- [6] ITRS. *International Technology Roadmap For Semiconductors - 2006 Edition*. Semiconductor Industry Association, 2006.
- [7] J. Lawrence Carter and Mark N. Wegman. Universal Classes of Hash Functions. *Journal of Computer and System Sciences*, 18:143–154, 1978.
- [8] C. M. Jeffery, A. Basagalar, and R. J. O. Figueiredo. Dynamic Sparing and Error Correction Techniques for Fault Tolerance in Nanoscale Memory Structures. In *4th IEEE Conference on Nanotechnology*, 2004.
- [9] P. J. Kuekes, W. Robinett, G. Seroussi, and R. S. Williams. Defect-Tolerant Interconnect to Nanoelectronic Circuits: Internally Redundant Demultiplexers Based on Error-Correcting Codes. *Journal of Nanotechnology*, 16:869–882, June 2005.
- [10] S. Lin and D. J. Costello. *Error Control Coding, Second Edition*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2004.
- [11] M. Mishra and S. Goldstein. Defect Tolerance at the End of the Roadmap. In *ITC*, pages 1201–1211, 2003.
- [12] J. V. Neuman. Probabilistic Logic and the Synthesis of Reliable Organisms from Unreliable Components. *Automata Series, Editors: C. Shannon and J. McCarthy, Princeton Univ. Press*, pages 43–98, 1956.
- [13] E. Ou and W. Yang. Fast Error-Correcting Circuits for Fault-Tolerant Memory. In *MTDT*, pages 8–12, 2004.
- [14] M. Ramakrishna, E. Fu, and E. Bahcekapili. Efficient Hardware Hashing Functions for High Performance Computers. *IEEE Transactions on Computers*, 48(12):1378–1381, 1997.
- [15] J. S. Reeve and K. Amarasinghe. A parallel Viterbi Decoder for Block Cyclic and Convolution Codes. *Journal of Signal Processing*, 86(2):273–278, 2006.
- [16] F. Sun and T. Zhang. Two Fault Tolerance Design Approaches for Hybrid CMOS/Nanodevice Digital Memories. In *IEEE International Workshop on Defect and Fault Tolerant Nanoscale Architectures (Nanoarch)*, 2006.
- [17] M. B. Tahoori. A Mapping Algorithm for Defect-Tolerance of Reconfigurable Nano-Architectures. In *ICCAD '05: Proceedings of the 2005 IEEE/ACM International conference on Computer-aided design*, pages 668–672.
- [18] J. Vollrath, U. Lederer, and T. Hladshchik. Compressed Bit Fail Maps for Memory Fail Pattern Classification. *Journal of Electronic Testing*, 17(3-4):291–297, 2001.
- [19] G. Wang, W. Gong, and R. Kastner. Defect-Tolerant Nanocomputing Using Bloom Filters. In *ICCAD 2006*, November 2006.
- [20] E. Witchel, J. Cates, and K. Asanović. Mondrian Memory Protection. In *Proceedings of ASPLOS-X*, Oct 2002.
- [21] M. Ziegler and M. Stan. CMOS/nano Co-Design for Crossbar-Based Molecular Electronic Systems. *IEEE Transactions on Nanotechnology*, 2:217–230, 2003.