The NanoBox Project: Exploring Fabrics of Self-Correcting Logic Blocks for High Defect Rate Molecular Device Technologies

AJ KleinOsowski David J. Lilja
ajko@ece.umn.edu lilja@ece.umn.edu
Department of Electrical and Computer Engineering
Digital Technology Center, University of Minnesota, Minneapolis, MN 55455

Abstract

Trends indicate that emerging process technologies, including molecular computing, will experience an increase in the number of noise induced errors and device defects. In this paper, we introduce the NanoBox, a logic lookup table with fault tolerance coding applied to the lookup table bit string. In this way, we contain and self-correct errors within the lookup table, thereby presenting a robust logic block to higher levels of logic design. We explore five different NanoBox coding techniques. We also examine the cost of implementing two different circuit blocks using a homogenous fabric of NanoBox logic elements: 1) a floating point control unit from the IBM Power4 microprocessor and 2) a four-instruction ALU. In this initial investigation, our results are not meant to draw definitive conclusions about any specific NanoBox implementation, but rather to spur discussion and explore the feasibility of fine-grained error correction techniques in molecular computing systems.

1 Introduction

Recent work in physics, chemistry, and materials science has produced nanometer-scale structures out of exotic materials using sophisticated fabrication techniques [7, 11, 15]. These new devices have the potential to be the "killer device" for the next generation of computers. However, there is widespread agreement among device researchers that nanodevices will have much higher manufacturing defect rates, very low current drive capabilities, and much more sensitivity to noise-induced errors [2, 4, 8]. The key advantage of nanotechnology devices is their small size and the resulting unprecedented level of integration expected in designs constructed with these devices.

As contemporary CMOS devices scale down and multi-gigahertz designs emerge, circuit topologies must change to account for shorter wires and higher densities of noise-induced errors [13]. Manufacturing flawless chips will become prohibitively expensive, if not impossible. Instead of assuming that defects and transient errors are uncommon, future circuits must adapt to, and coexist with, the substantial numbers of manufacturing defects and high transient error rates.

In this work, we introduce the NanoBox, a self-correcting logic block for addressing the increasing error densities in emerging process technologies. The NanoBox consists of a logic lookup table with appropriate error detection and correction. Due to the scaling of emerging process technologies, and the resulting increase in transistor budgets, the area overhead of adding bit-level fault-tolerance to circuits may now be feasible.

With self-correcting logic blocks, the error coverage of circuit designs can be increased, invisibly to the logic designer. Existing hardware description language code can be synthesized to these self-correcting logic blocks using existing place and route techniques. By correcting errors at the fine-grained bit-level, designs may not need complicated module or system level redundancy. Reducing, and perhaps eliminating, module and system level redundancy will decrease the complexity of designs, making them easier to verify. Ultimately, shifting to fine-grained, self-correcting logic blocks aims to leverage existing intellectual property in logic design, enabling the transition of these existing designs to emerging fabrication technologies with minimal effort.

2 NanoBox Circuit Topology

The following sections walk through the overall NanoBox concept, a NanoBox implementation using triple modular redundancy, and a NanoBox implementation using information coding.

2.1 The NanoBox Concept

To demonstrate how the NanoBox approach could be used in combinational logic design, we present an example using a 4-bit sum operation. Figure 1(a) shows a sum function of four variables as it would be constructed using conventional logic elements. Figure 1(b) shows the same function constructed with an encoded lookup table. The function inputs are fed to a decoder which addresses an array of memory cells. The value of the addressed memory cell is fed to a sense amplifier which is used as the function output.

In each lookup table, extra memory cells are added for the check bits that encode an error correction code of the function truth table bits.

We envision that these NanoBox circuit elements would be created from hardware description language code by the synthesis step of computer aided design (CAD) tools. The synthesis step would determine the contents, 1 or 0, of each truth table memory cell, as well as the contents of each check bit cell.

During normal circuit operation, the contents of the memory cells do not change. Under transient fault conditions, or in the presence of manufacturing defects, the contents of the memory cells may be incorrect. Additionally, there may be noise on the wires within the NanoBox which result in a memory cell which appears to have an incorrect value.

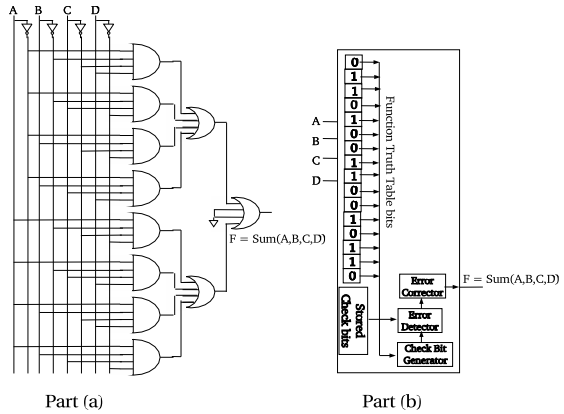

Figure 1. (a) An example combinational logic circuit constructed with conventional devices. (b) The same logic circuit constructed with an error-correcting lookup table.

Whenever the inputs to the lookup table change, the truth table bits are fed into the check bit generator, which recalculates the check bits. These newly calculated check bits are then compared with the stored check bits in the error detector. The results of the error detector are fed into the error corrector, which makes changes to any flipped bits at the function output. The corrected function output is then used as the actual output of the lookup table.

A point to note is that we do not change the stored value of the memory cells. We assume that the errors are either transient and will return to their correct value, or the errors are static and cannot be corrected. Our NanoBox concept aims to identify and correct errors at the function output, not to correct the stored values within the function truth table.

Depending on the characteristics of the devices used to implement the NanoBox, the combinational logic needed for the check bit generator, error detector, and error corrector may also be implemented with NanoBoxes. For example, a very large NanoBox could use information coding on the function truth table, and then the error correction circuitry within the information coding NanoBox could be implemented using smaller, triple modular redundancy NanoBoxes. Triple modular redundancy and information coding NanoBoxes are explained in the following sections.

2.2 Triple Modular Redundancy NanoBox

The triple modular redundancy NanoBox implementation has three copies of the memory cell array. Each of the three copies stores an identical copy of the function truth table. The error detection and correction circuitry consists of a simple three input majority gate. Whenever the NanoBox inputs change, the three copies of the memory cell addressed by the NanoBox inputs are fed to the majority gate. The output of the majority gate is used as the NanoBox output. As long as two out of the three copies of the memory cell being addressed have the correct value, the NanoBox will produce the correct function output. All of the memory cells not addressed by the NanoBox inputs may be in error without affecting the NanoBox output since the non-addressed memory cells are not observed by the majority gate.

2.3 Information Coding NanoBox

In the information coding NanoBox implementation a small number of memory cells are added to store check bits which encode a function of the truth table bits. The number of check bits varies based on the size of the truth table and the coding used. In this evaluation, we use 16-bit truth tables. We explore three different types of information coding: Hamming code, Hsiao code, and Reed Solomon code [10]. Hamming codes are typically used in microprocessor buses or on short memory words. Hsiao codes are a derivative of Hamming codes, with a modified parity check matrix. Each check bit has the same number of data bits fed into it's regeneration logic, causing all of the regenerated check bits to arrive at the same time. Reed Solomon codes are typically used in memory systems with long memory words. The long words are broken into symbols and the code allows for all of the bits in a symbol to be reconstructed in the presence of an error. In this way, a multi-bit burst error can be corrected.

Whenever the NanoBox inputs change, all of the function truth table bits are fed to the check bit generator, which recalculates the check bits based on the current, and perhaps faulty, stored truth table bits. These recalculated bits are then fed to the error detector which compares the recalculated bits to the stored, and perhaps faulty, check bits. The resulting syndrome identifies which stored bit, if any, is in error. If the error is on the memory cell being addressed by the NanoBox inputs, the memory cell value is inverted before being used as the NanoBox output. If the error is on a memory cell not addressed by the NanoBox inputs, the error is ignored.

The information coding NanoBox has relatively small overhead in terms of extra memory cells. However, the error detector and error corrector are moderately complex and require significant area. As mentioned in Section 2.1, in a large information coding NanoBox, the error detector and error corrector could be implemented with other, smaller and simpler, NanoBoxes.

3 Evaluation Methodology

Our evaluation set out to explore the feasibility of mapping existing microprocessor designs to NanoBox encoded lookup tables. We also wanted to gather preliminary information about what coding techniques are best suited for our NanoBox encoded lookup tables.

We began our NanoBox evaluation by constructing SPICE models of the encoded lookup tables. Since molecular nanotechnology devices are not yet mature enough to have specific area, power, and timing models, we constructed our lookup tables from contemporary CMOS devices. Also, for this initial investigation, the check bit generator, error detector, and error corrector are implemented with full custom CMOS, rather than with additional, smaller, NanoBox circuit elements. The VLSI schematics of our encoded lookup tables will undoubtedly change based on the specifics of a particular emerging technology device. By using CMOS devices, though, we are able to explore proof-of-concept simulations and begin evaluating the relative area, power, and timing trade-offs of the different coding techniques.

Our simulations used the Cadence software tools suite to develop our schematics and extract SPICE netlists [1]. We used BSIM compact models for a 90nm silicon on insulator process for our SPICE simulations. In all simulations except the synthesized CMOS floating point control unit, NMOS transistors were sized at 4λ and PMOS transistors were sized at 6λ.

We implement two circuit blocks with NanoBoxes. The first block is the floating point unit control circuit for the IBM Power4 microprocessor [14]. Since this floating point control circuit block contains millions of nodes, we also implement a much smaller block, a four-instruction arithmetic logic unit (ALU). In the floating point control unit analysis, we focus on how well existing microprocessor designs map to NanoBox logic elements. Our analysis of the ALU focuses more on the tradeoffs between different NanoBox logic element implementations.

3.1 Floating Point Control Unit Evaluation Methodology

We examine the floating point unit control circuit for the IBM Power4 microprocessor. This circuit block was chosen due to the irregular physical design tendencies of control logic. Also, memory circuits are well covered by traditional error correcting code techniques and datapath circuits are well covered by traditional modular or time redundancy fault tolerant techniques. Error correction in control logic, in contrast, has not been well explored.

We investigate how well the floating point control circuit VHDL code maps to a fabric of NanoBoxes, in terms of memory cell usage and wasted area. In this initial investigation, we assume a homogenous fabric of four-input, 16-bit NanoBoxes. This size lookup table was chosen due to its common use in contemporary field programmable gate array (FPGA) systems [6]. We use the Xilinx Foundations Series ISE [5] software to map our floating point control unit VHDL code to lookup tables. This software reads in VHDL code, breaks the code into functions of four (or fewer) inputs, then determines the connections between the lookup tables.

The control circuit block contains millions of nodes and therefore it is infeasible to simulate it in its entirety with a SPICE model. Therefore, we choose one of the pipeline control stages and develop SPICE netlists of this one pipeline stage. Even this simplified netlist contains hundreds of thousands of nodes, so we simulate using only nominal device sizes, temperature, and voltages.

We simulate six different versions of the single floating point pipeline control circuit: 1)*cmos synth*–The netlist developed from a fully automated physical design synthesis, with varying device widths; 2)*cmos*–The synthesized netlist, converted to only minimum size devices; 3)*nocode*–The netlist mapped to NanoBoxes with no error correction coding; 4)*tmr*–The netlist mapped to NanoBoxes with triple module redundancy coding; 5)*hamming*–The netlist mapped to NanoBoxes with Hamming code information coding; and 6)*hsiao*–The netlist mapped to NanoBoxes with Hsiao code information coding. We chose to convert the synthesized netlist to minimum size devices due to the trend toward homogeneously sized nanodevices [3, 9, 12, 16].

We use an automated test pattern generator (ATPG) tool to create stimulus vectors for the floating point control logic module. We simulate 35 different combinations of primary inputs. These primary input combinations are random and do not necessarily reflect realistic module input. Instead, they force activity in the circuit and allow us to test for functional correctness, average power consumption, and worst case delay.

3.2 Arithmetic Logic Unit (ALU) Evaluation Methodology

Our second logic module is a four-instruction arithmetic logic unit (ALU). Although datapath logic may be better suited to course-grained modular or time redundancy techniques, our ALU presents us with a well-defined, small circuit block for detailed analysis.

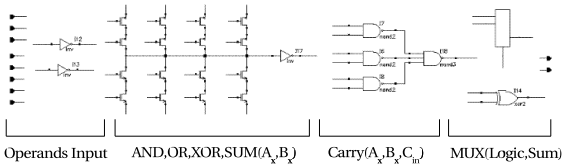

Figure 2. Bit slice of the four instruction ALU, constructed with full custom CMOS devices.

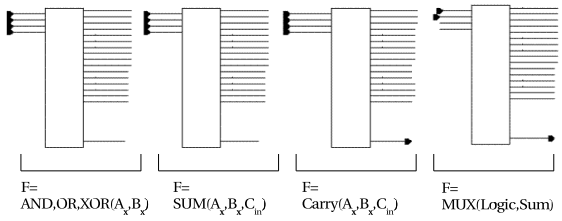

Figure 3. Bit slice of a four instruction ALU from Figure 2, constructed with encoded lookup table NanoBoxes.

Our ALU is a smaller circuit than our floating point control circuit, so we simulate the ALU using Monte Carlo analysis. In this way, we model device instability and manufacturing process variations, and are able to evaluate the error coverage of the different NanoBox coding techniques. We run ten cases in our Monte Carlo analysis. In each case, every device has a randomly chosen length, width, and threshold voltage. The device lengths vary +/- 0 to 17 percent from the nominal value, the device widths vary +/- 0 to 31 percent from the nominal value, and the device threshold voltages vary +/- 0 to 46 percent from the nominal value. The results we discuss in Section 4.2 are the statistical summary from the ten Monte Carlo runs.

We simulate six different versions of the ALU: 1)*cmos*–The netlist developed from a full custom CMOS ALU made of minimum size devices, as shown in Figure 2; 2)*nocode*–The ALU mapped to NanoBoxes with no error correction coding; 3)*tmr*–The ALU mapped to NanoBoxes with triple module redundancy coding; 4)*hamming*–The ALU mapped to NanoBoxes with Hamming code information coding; 5)*hsiao*–The ALU mapped to NanoBoxes with Hsiao code information coding; and 6)*reed-solomon*–The ALU mapped to NanoBoxes with Reed Solomon code information coding. Each bit slice of the ALU uses four NanoBoxes, as shown in Figure 3.

Given a three bit opcode, the ALU calculates the AND, OR, XOR, and ADD of two bits. As the stimulus for our ALU, we perform sixteen computations. We calculate each of the four ALU instructions, AND, OR, XOR, and ADD, with each of the four 2-bit input combinations. For each instruction, we cycle through the input combinations 00, 01, 10, 11. In this way, we exercise all of the computations possible with our ALU.

4 Results

The following sections analyze the NanoBox implementations of a floating point unit control circuit and a four-instruction arithmetic logic unit (ALU), described in Section 3.

4.1 Floating Point Control Circuit Results

Table 1 shows the mapping of the nine modules within the floating point control unit to lookup tables. From this data, we see that a significant portion of the floating point control circuit VHDL code could not map into four-input lookup tables. Since we are using only four-input lookup tables in our homogenous fabric of NanoBoxes, logic which mapped into a two-input or a three-input lookup table was implemented with a four-input lookup table with one or two of the inputs tied to ground. Tying these inputs to ground results in lookup table memory cells which are not used. The last column of Table 1 shows the fraction of the lookup table memory cells used. We see that the average usage is 66 percent, with a maximum usage of 78 percent and a minimum usage of 46 percent.

Our NanoBox overhead decreases as we increase the size of the lookup tables. However, the mapping in Table 1 shows that even four-input lookup tables are, in many cases, too large. Detailed analysis of the mapping output showed that control logic is typically IO bound. This means the control logic consists mainly of many functions of few variables, rather than few functions of many variables. Module 6 from Table 1, with 19 lookup tables, is used in the SPICE models which give us area, power, and delay results, shown in Figures 4, 5, and 6, respectively.

Figure 4 shows the area of our floating point pipeline control circuit implementations. These results indicate an exponential increase in the amount of area as we move from a lookup table implementation with no error correction codes, to lookup table implementations with complex error correction codes. This result is somewhat intuitive, since more complex error correction codes require a more complex error detector and error corrector.

Although the area overheads are startling for CMOS devices, our NanoBox technique may be feasible for non-silicon device technologies, given the expected unprecedented increases in device density projected with these technologies [3, 9, 12, 16].

Figure 5 shows the average power consumption of our floating point pipeline control circuit implementations. The average power increases sharply (122x) when moving from a CMOS implementation to a lookup table implementation. The choice of a specific lookup table coding technique is comparatively insignificant in terms of average power.

Figure 6 shows the worst case delay of our floating point pipeline control circuit implementations. Similar to the average power results, delay increases sharply (6x) when moving from a CMOS implementation to a lookup table implementation. The choice of lookup table coding technique is comparatively insignificant in terms of delay. Surprisingly, the implementation with triplicated memory bits (tmr) showed the worst delay. We attribute this delay to the increased gate capacitance between the decoder and the three copies of the memory cell array, and to the fact that we did not model manufacturing variations and device faults in the floating point pipeline control SPICE simulations. (Manufacturing variations and device faults are modeled with our Monte Carlo SPICE simulations for the ALU circuit block in Section 4.2.) If we had modeled wire capacitance or device switching

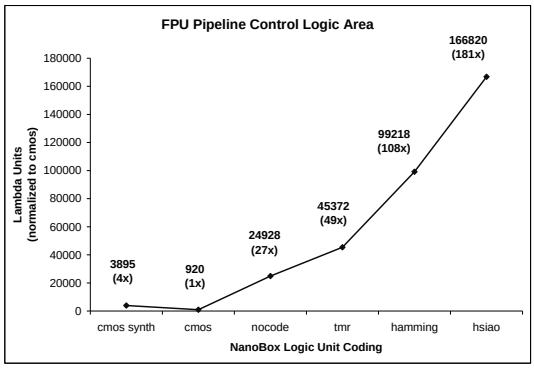

Figure 4. Area of different implementations of the floating point control logic.

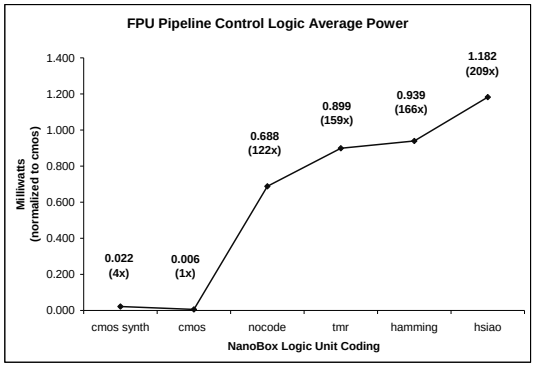

Figure 5. Average power consumption of different implementations of the floating point control logic.

in the error corrector or error detector, we suspect the information coding implementations would show more delay.

4.2 Arithmetic Logic Unit (ALU) Results

Figure 7 shows the area of our ALU implementations. Similar to the area results for the floating point pipeline control unit, the area increases exponentially as we move from a lookup table implementation with no error correction, to a lookup table implementation with a complex error correction technique. The overhead of the Reed Solomon code implementation has especially high overhead, with a 391x increase over the area of the CMOS implementation.

Figure 8 shows the mean average power consumption over ten Monte Carlo runs of each of the ALU implementations. In each run, each device has a randomly chosen length, width, and threshold voltage, as described in Section 3.2. Figure 8 also plots the coefficient of variation, which is the ratio of the standard deviation to the mean average power. The results in Figure 8 show a significant (83x) increase in power when moving from a CMOS implementation to a lookup table implementation. The power consumption increases linearly as we use more complex error correction codes in the lookup tables. The coefficient of variation drops as we move from a CMOS implementation to a lookup table implementation, but then rises again as we move to more complex error correction codes. Not surprisingly, the nocode,

Table 1. Lookup table (LUT) mapping of floating point control unit circuit modules. The *Memory Cells with LUT4s Only* column shows how many total lookup table memory cells are available when using only four-input lookup tables. The *Memory Cells with LUT4, LUT3, LUT2* column shows how many of these lookup table memory cells are used by the VHDL mapping. The *Cell Useage* column calculates the percent usage of lookup table memory cells with the VHDL to lookup table mapping. (*) Module 6 from this table, with 19 lookup tables, is used in the SPICE models to determine area, power, and delay results.

FPU Pipeline Control Logic Lookup Table Memory Cell Usage						
	LUT4	LUT3	LUT2	Memory Cells with LUT4s Only	Memory Cells Used with LUT4s, LUT3s, and LUT2s	Cell Usage
module 1	82	39	22	2288	1712	75%
module 2	29	14	9	832	612	74%
module 3	23	15	10	768	528	69%
module 4	65	30	28	1968	1392	71%
module 5	76	22	21	1904	1476	78%
(*)module 6	9	4	6	304	200	66%
module 7	36	7	97	2240	1020	46%
module 8	54	46	33	2128	1364	64%
module 9	17	13	22	832	464	56%

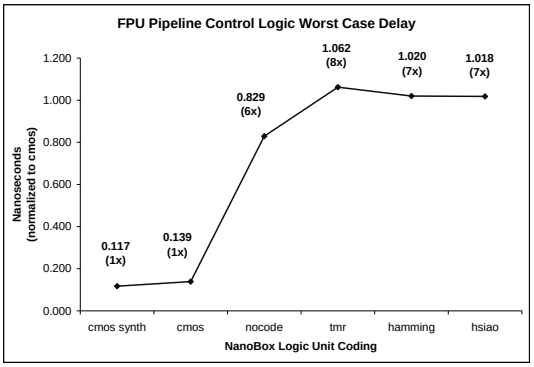

Figure 6. Worst case delay of different implementations of floating point control logic.

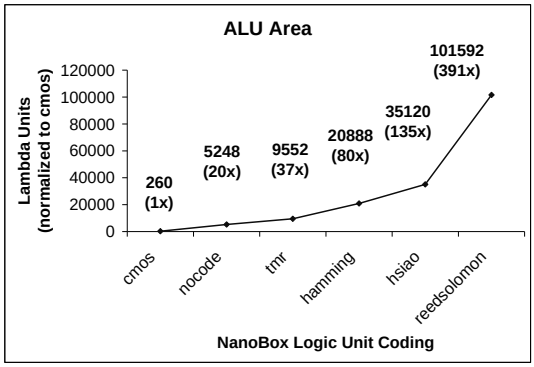

Figure 7. Area of different implementations of the four-instruction ALU.

tmr, and hsiao implementations, all of which have fairly regular physical design, have a low coefficient of variation.

Figure 9 shows the mean worst case delay and coefficient of variation over ten Monte Carlo runs of each of the ALU implementations. The results in Figure 9 show a significant (4x) increase in delay when moving from a CMOS implementation to a lookup table implementation. The delay overhead is constant at 5x for all coding techniques. The coefficient of variation is also fairly constant across all implementations, ranging from a minimum of 0.0396 for the Hsiao implementation to a maximum of 0.0860 for the Hamming implementation.

5 Future Work

The work described in this paper is an initial investigation into developing self-correcting combinational logic circuits using encoded logic lookup tables. Our foremost future work is to investigate ways to implement the error detection and correction logic using additional lookup tables, rather than with conventional CMOS circuitry. We also plan to investigate ways to simplify the correction and detection logic, such as subsetting the lookup table or developing different correction codes.

To extend the NanoBox approach, we are developing the Recursive NanoBox Processor Grid using the NanoBox circuit elements. This highly parallel, application-specific processor architecture is being used to evaluate whether the self-correcting logic blocks provide adequate error coverage over an entire microarchitecture, or whether modular and system level fault tolerance techniques still need to be used in conjunction with the fine-grained fault tolerance of the NanoBoxes.

6 Conclusion

The yield, reliability, and error characteristics of new device technologies are expected to be substantially different from the corresponding characteristics of conventional CMOS devices. Existing fault tolerance techniques used in microprocessors assume relatively low manufacturing defect densities and infrequent dynamic faults. Projecting forward, it is expected that future circuit topologies will have substantially higher defect densities and dynamic faults. These differences between current and future devices will require fundamentally new design techniques in which microprocessors are designed from the start to coexist with many defects and high densities of transient errors.

We introduce The NanoBox, a logic lookup table which uses

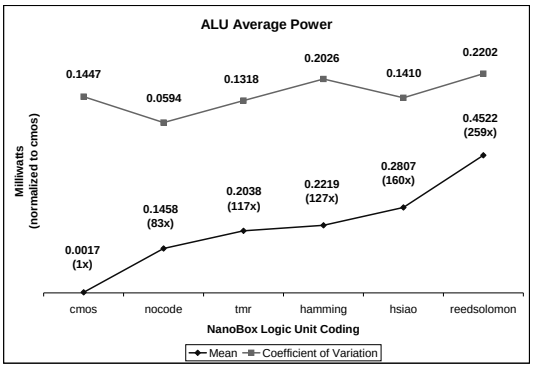

Figure 8. Mean average power and coefficient of variation of different implementations of the four-instruction ALU, over ten cases of a Monte Carlo analysis.

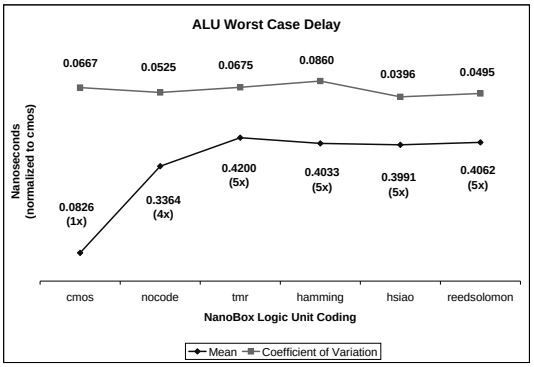

Figure 9. Mean worst case delay and coefficient of variation of different implementations of the four-instruction ALU, over ten cases of a Monte Carlo analysis.

error correction on the function truth table, thereby achieving fine-grained fault tolerance. These encoded lookup tables are able to dynamically correct faults within the lookup table, thereby presenting a robust logic block to higher levels of circuit integration and microarchitecture. Standard hardware description language code can be mapped to these NanoBox circuit elements using existing synthesis techniques. Logic designers are therefore able to increase the error coverage of their designs without adding complex modular or system level fault tolerance techniques.

Our results show preliminary indications that the best fine-grained resilience to device faults and variations is achieved by balanced error correction codes, such as a triplicated memory array or a Hsiao information code derivative. These designs have a fairly regular physical design and consistent length critical path. However, it should be noted that implementing circuits with homogeneous fabrics of logic elements incurs a significant area, power and delay overhead. Some of this overhead may be reduced if circuit structures can be reorganized into few functions of many variables, rather than many functions of few variables. If circuit designs can tolerate overheads in the range presented in this work, our NanoBox approach of using error correction codes on lookup table memory strings shows promise for the realm of molecular computing.

7 Acknowledgements

The authors would like to thank the University of Minnesota NanoBox Research group for their input during discussions related to this work: Vijay Rangarajan, Priyadarshini Ranganath, Kevin KleinOsowski, Mahesh Subramony, Chris Hescott, Drew Ness, and Professor Richard Kiehl. The authors would also like to thank the IBM Corporation for their extensive support of this project. In particular, we would like to thank Kevin Nowka for his graduate internship funding of the first author, Marty Schmookler for his floating point unit VHDL code, Richard Williams for his support of experimental device compact models, Tim Dell for his assistance with information codes, and the LoadLeveler support team for their computational resources. This project was also supported by NSF grant number CCR-0210197, the University of Minnesota Digital Technology Center, and the Minnesota Supercomputing Institute.

References

[1] W. Banzhaf. *Computer-Aided Circuit Analysis Using SPICE*. Prentice Hall, 1989.

[2] S. R. Corporation. International technology roadmap for semiconductors (ITRS). Document available at `http://public.itrs.net`, 2001.

[3] S. R. Corporation. International technology roadmap for semiconductors (ITRS) 2002 update. Document available at `http://public.itrs.net`, 2002.

[4] S. R. Corporation. SRC research needs document for 2002-2007. Document available at `http://www.src.org/fr/current_calls.asp`, June 2002.

[5] X. Corporation. Xilinx ISE Foundation Software. a programmable logic design environment. Product information available at `http://www.xilinx.com/`.

[6] X. Corporation. Virtex-II Pro Platform FPGAs: Functional description. Document available at `http://www.xilinx.com/`, September 2002.

[7] L. Geppert. The amazing vanishing transistor act. *IEEE Spectrum*, pages 28–33, October 2002.

[8] H. Iwamura, M. Akazawa, and Y. Amemiya. Single-electron majority logic circuits. *IEICE Transactions on Electronics*, E81-C(1):42–48, 1998.

[9] P. Kornilovitch, A. Bratkovsky, and R. S. Williams. Bistable molecular conductors with a field-switchable dipole group. *Physical Review B*, 66(245413):713–715, December 2002.

[10] P. K. Lala. *Self-Checking and Fault-Tolerant Digital Design*. Morgan Kaufmann, 2001.

[11] J. Markoff. Electronic memory research that dwarfs the silicon chip. *The New York Times*, October 20 2003.

[12] N. A. Melosh, A. Boukai, F. Diana, B. Geradot, A. Badolato, P. M. Petrof, and J. R. Heath. Ultrahigh-density nanowire lattices and circuits. *Science*, March 2003.

[13] P. Shivakumar, M. Kistler, S. W. Keckler, D. Burger, and L. Alvisi. Modeling the effect of technology trends on the soft error rate of combinational logic. In *International Conference on Dependable Systems and Networks (DSN)*, 2002.

[14] J. M. Tendler, J. S. Dodson, J. J. S. Fields, H. Le, and B. Sinharoy. Power4 system microarchitecture. *IBM Journal of Research and Development*, 46(1):5–25, 2002.

[15] K. L. Wang. Issues of nanoelectronics: A possible roadmap. *Journal of Nanoscience and Nanotechnology*, 2(3/4):235–266, 2002.

[16] S. Xiao, F. Liu, A. E. Rosen, J. F. Hainfeld, N. C. Seeman, K. Musier-Forsyth, and R. A. Kiehl. Assembly of nanoparticle arrays by DNA scaffolding. *Journal of Nanoparticle Research*, 4(4):313–317, 2002.