The Recursive NanoBox Processor Grid: A Reliable System Architecture for Unreliable Nanotechnology Devices

AJ KleinOsowski[*] ajko@mail.ece.umn.edu
Kevin KleinOsowski kevinko@mail.ece.umn.edu
Vijay Rangarajan rvijay@ece.umn.edu

Priyadarshini Ranganath priya@ece.umn.edu
David J. Lilja lilja@ece.umn.edu

Department of Electrical and Computer Engineering
Digital Technology Center, University of Minnesota
200 Union Street SE, Minneapolis, MN 55455
Telephone: 612-625-5007 Fax: 612-625-4583

Abstract

Advanced molecular nanotechnology devices are expected to have exceedingly high transient fault rates and large numbers of inherent device defects compared to conventional CMOS devices. Instead of trying to manufacture defect-free chips in which transient errors are assumed to be uncommon, future processor architectures must be designed to adapt to, and coexist with, substantial numbers of manufacturing defects and high transient error rates. We introduce the Recursive NanoBox Processor Grid as an application specific, fault-tolerant, parallel computing system designed for fabrication with unreliable nanotechnology devices. This architecture is composed of many simple processor cells connected together in a two dimensional grid. It uses a recursive black box architecture approach to isolate and mask these transient faults and defects. In this initial study we construct VHDL models of one processor cell and evaluate the effectiveness of our recursive fault masking approach in the presence of random transient errors. Our analysis shows that this architecture can calculate correctly 100 percent of the time with raw FIT (failures in time) rates as high as 10^{23}*, while computing correctly 98 percent of the time when experiencing raw FIT rates in excess of* 10^{24}*, which is twenty orders of magnitude higher than the FIT rates of contemporary CMOS device technologies. We achieve this error correction with an area overhead on the order of 9x, which is quite reasonable given the high integration densities expected with nanodevices.*

Keywords: nanotechnology, microarchitecture, fault-masking, fault-injection, VLSI
Submission Category: Regular Paper, Dependability in VLSI

Approximate Word Count: 6700

[*]Corresponding Author. This draft is under review for the 2004 International Conference on Dependable Systems and Networks.

1 Introduction

Advances in physics, chemistry, and materials science have produced nanometer-scale structures out of exotic materials using sophisticated fabrication techniques [3, 11, 35]. 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, compared to conventional CMOS devices, nanodevices will have much higher manufacturing defect rates, they will have much lower current drive capabilities, and they will be much more sensitive to noise-induced errors [5, 6, 7, 15]. The key advantage to nanotechnology devices is their small size and the resulting unprecedented level of integration in designs constructed with these devices.

In addition to adapting to non-silicon device technologies, circuit designs using contemporary CMOS devices must change in the near future to account for shorter wires and higher densities of noise-induced errors as these devices are scaled down and multi-gigahertz designs emerge [30]. 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.

The main objective of the *Recursive NanoBox* project is to develop circuit and architecture-level techniques to tolerate the expected defect densities and noise sensitivities of new nanotechnology devices. In this paper, we introduce the NanoBox concept and evaluate different implementations of this noise tolerant processor architecture approach using fault-injection experiments.

The term *NanoBox* refers to a black box entity that uses a specified fault-tolerance technique. The NanoBox Processor Grid is a hierarchy of these black boxes, where a different fault-tolerance technique can be used at the bit, module, and system levels. At the bit-level, we use space or information redundancy by adding error correction to the truth table of a field programmable gate array (FPGA)-style lookup table [8, 16]. At the module level, we use space or time redundancy by having multiple copies of the ALU within the processor cell, or by having the ALU compute each instruction multiple times. At the system level, we use space redundancy with a grid of identical processor cells working together on a parallel computation. With this box-within-a-box approach, faults not correctable at one level of the hierarchy should be covered by the fault tolerance technique of a box at a higher level of the hierarchy. For example, if, during a computation, a lookup table experiences more errors than it can correct with its code bits, the error should be detectable by the space or time redundancy mechanisms implemented at the module level. These higher levels of redundancy should allow the computation to produce correct results in spite of uncorrectable errors in lower levels of the hierarchy.

We begin in Section 2 by describing our recursive approach to obtaining high reliability for systems fabricated with nanotechnology devices. Section 3 then describes the overall NanoBox architecture and the detailed operation of each processor cell. Our methodology for evaluating the fault tolerance capabilities of the pro-

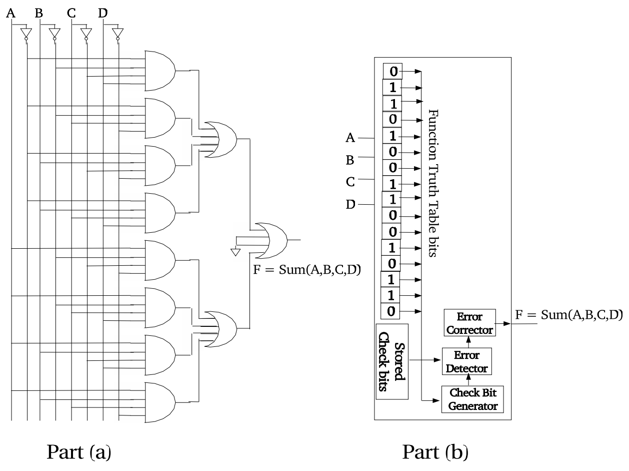

Figure 1: (a) An example combinational logic circuit constructed with conventional devices. (b) The same logic circuit constructed with an error-correcting lookup table. Each bit of the logic function truth table, along with the truth table check bits, is stored in a memory cell.

cessor cells is described in Section 4, while the results of this evaluation are discussed in Section 5. Section 6 contrasts our approach to related work in fault tolerance, followed by some suggestions for future work in Section 7. We conclude our analysis of the NanoBox architecture in Section 8.

2 Recursive Approach to Reliability

One of the novel aspects of our proposed NanoBox processor comes from its multiple levels of fault isolation and dynamic fault correction. In the paragraphs below, we describe each of the three levels of fault isolation and dynamic fault correction.

2.1 Bit Level Fault Tolerance

At the bit level, we use field programmable gate array (FPGA)-style lookup tables to implement the desired logic. These lookup tables contain error correction codes which can dynamically detect and, depending on the error densities and codes used, actually correct errors.

To demonstrate how this approach could be implemented, 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. In each lookup table, extra check bits are added that encode an error correction code of the bits that comprise the logic function.

Whenever the lookup table is accessed, 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 in the function output. The corrected function output then is used as the actual output of the lookup table. An analysis of tradeoffs among different lookup table coding techniques was conducted in [16, 17].

The lookup table check bits and error detector and corrector will vary considerably depending on what coding technique is used. For example, an information code (i.e. Hamming, Hsiao, Reed-Solomon, etc. [18]) lookup table uses a small number of additional memory cells to store check bits. The number of check bits varies based on the size of the truth table and the coding used. In general, though, the number of check bits is small compared to the number of bits in the lookup table bit string. The error detector and corrector for an information code, however, are moderately complex and require significant area. In a large information coding lookup table, the error detector and error corrector could be implemented with other, smaller and simpler, lookup tables.

A triple modular redundancy lookup table uses two additional copies of the memory cell array as check bits instead of using a mathematical function of the bits. The error detector and corrector for this configuration consists of a simple three input majority gate. Unlike an information code lookup table, a triple modular redundancy code lookup table has high check bit overhead, but low error detector and corrector overhead. The associated trade-offs between check bit overhead and the required supporting circuitry have been explored in [16, 17].

2.2 Module Level Fault Tolerance

At the next level of the hierarchy, in the processor cell, simple ALUs, each with a small read/writable memory, are constructed from the fault-tolerant lookup tables. These ALU's use space or time redundancy to dynamically check for errors which may have occurred during the instruction execution. Each instruction is executed multiple times, either concurrently using multiple ALUs, or serially using a time-redundant ALU. The repeated results are fed into a voter circuit which determines the final result of the instruction.

Our read/writable memory may have single-event upsets causing transient bit flips. To combat these errors, critical fields within the memory word are stored in triplicate. Whenever these critical fields are accessed, the majority value of these triplicated fields is computed and that majority value is used as the value of the field. Contemporary information coding techniques could also be used on the memory words, for additional error coverage.

2.3 System Level Fault Tolerance

At the top-most level of the hierarchy, entire processor cells may be disabled if they exceed a predefined error threshold. A heartbeat signal, generated within the processor cell, is used to determine if the cell is still active. A watchdog unit in the communication fabric monitors these processor cell heartbeat signals and determines if a cell has exceeded its error threshold. If a processor cell is disabled, the communication fabric surrounding the disabled processor cell will cease sending instructions to the that processor cell. If the router and cell memory are still functioning, the contents of the cell memory will be sent to the surrounding processor cells so that they can finish any outstanding computations.

3 The Recursive NanoBox Processor System Architecture

We introduce the Recursive NanoBox Processor cell architecture to evaluate the hierarchical approach to reliability presented in the previous section. This architecture is a nanocomputing system that works as an application specific co-processor to a conventional CMOS microprocessor. Multiple NanoBox Processor Grids, each designed for a different application, could be included with, and managed by, a single general purpose CMOS control processor. The control microprocessor packages data into a form the NanoBox Processor Grid understands, stores that data in its CMOS memory, then feeds the data to the NanoBox Processor Grid by a bus along one edge of the grid.

3.1 Architectural Overview

The NanoBox Processor Grid consists of a two-dimensional grid of processor cells, as shown in Figure 2. The exact number of processor cells is arbitrary, although we envision on the order of hundreds of processor cells in the NanoBox Grid. Each processor cell contains a simple ALU, a small amount of read/writable memory, and a communication router. Data traverses through the NanoBox Processor Grid using nearest neighbor communication among the processor cells. There are no cross-grid buses. Processor cells contain four 8-bit buses, with one bus connected to each of its neighbors. The processor cells along the right, left, and bottom edges have their outer edge bus disabled. Processor cells in the topmost row are connected to the conventional CMOS control processor by their outer edge bus. Through this top edge bus the control processor sends data to, and reads data from, the NanoBox Processor Grid.

Incoming communication and data packets are examined by the processor cell router and then either passed on to other cells, or routed to logic blocks within the processor cell. The processor cell IDs are laid out so that by looking at the destination ID within a data packet, a processor cell router can tell which direction, up, down, right, or left, to send the data packet. Moving away (down) from the control processor, row addresses decrease. Column addresses decrease while moving right in the processor grid.

The processor cells are composed of regularly patterned nanodevices that are surrounded by, and interconnected with, a coarse-grained grid of conventional CMOS circuits, as shown in Figure 3. These nanodevices can be used to construct logic functions using a lookup table approach, described in Section 2.1. Each lookup table will contain the truth table of some specific logic function. On the order of thousands of these lookup tables will be connected together to form a processor cell with appropriate functionality.

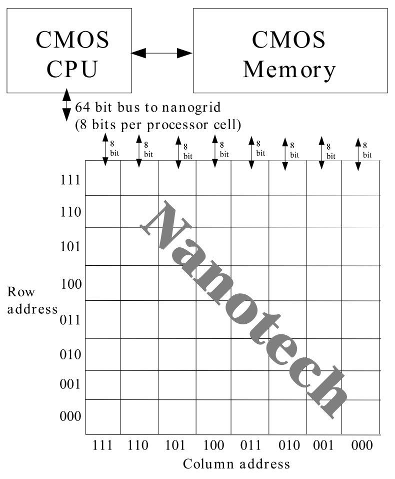

Figure 2: High-level view of the NanoBox Processor Grid showing row and column processor cell ID assignments. The 64-bit data bus (8 bits per processor cell) along the top edge of the grid is the only pin interface between the conventional CMOS control processor and the NanoBox Processor Grid.

3.2 Modes of Operation

Each processor cell has three mode signals, only one of which can be high at a time. These mode signal lines come from the control processor and are distributed to all processor cells. The three modes of operation are: *shift-in*, *compute*, and *shift-out*. All processor cells on the grid work concurrently and switch between modes at the same time.

3.2.1 Shift-In Mode

In shift-in mode, data packets are created by the off-grid control processor and fed to the NanoBox Processor Grid by way of the edge bus. These data packets contain a unique instruction ID, an ALU instruction, two operands, and the ID of the processor cell where the instruction will be computed.

The processor cell receives the start of a packet and then continues receiving data, 8 bits at a time, until the end of the packet is received. At that time, the processor cell examines the ID within the packet and determines if it should save the instruction and operands to its memory, or if it should pass the packet to one of its neighbors. The memory word which stores instructions and data is shown in Figure 4.

All processor cells stay in shift-in mode until the control processor finishes sending data to the edge bus. The control processor then waits for a specified number of cycles to ensure that all processor cells have received their data, after which it switches the processor cells to compute mode. The number of cycles the control processor must wait is determined by the number of processor cells in the NanoBox Grid.

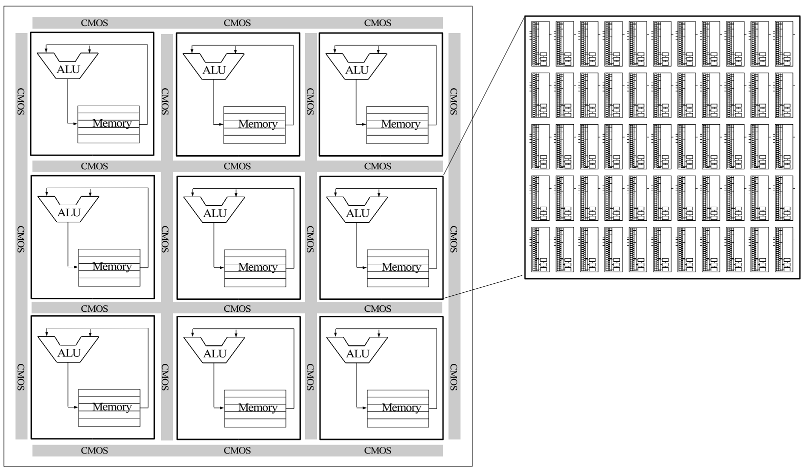

Figure 3: The architectural view of a nanocomputing system consists of a grid of regularly patterned nanodevices that implement lookup tables. Groupings of lookup tables form a processor-in-memory cell. The processor-in-memory cells are interconnected with a coarse-grained grid of conventional CMOS circuits.

6	3	3	8	3	8	8	8	8	8
Processor Cell ID	Data-Valid (triplicated)	To-Be-Computed (triplicated)	Unique Instruction ID	Instruction Opcode	Instruction Operand 1	Instruction Operand 2	Instruction Result 1	Instruction Result 2	Instruction Result 3

Figure 4: Bit fields used in the processor cell memory word.

3.2.2 Compute Mode

In compute mode, all of the processor cells cycle through their memories, reading one word at a time. Each memory word (shown in Figure 4) contains the operands and opcode for one ALU instruction. After the computation completes, the ALU control stores the memory word back to memory with the computation result. The `to-be-computed` bit then is cleared on that memory word. Three copies of the result are generated. These copies may be generated concurrently, using three separate ALUs (module-level space redundancy), or the results may be generated serially, using one ALU which computes the result three times (module-level time redundancy).

The ALU control loops through all of the memory words, computing each word of memory, storing results, and clearing the `to-be-computed` bit. If a processor cell fails during compute mode, the salvaged data from the failed processor cell may be transferred to nearby processor cells for computation. This data will appear in the nearby processor cell's memory as additional memory words with the `to-be-computed` bit set. Therefore, after the ALU control reaches the last memory word, it returns to the beginning of the memory and re-examines each word to see if the `to-be-computed` bit has been set. The ALU control continues to loop through the memory, word by word, as long as the processor cell is in compute mode.

3.2.3 Shift-out Mode

In shift-out mode, processor cells will assemble data packets from their memory words containing the unique instruction ID, and the majority vote of the three copies of the instruction result. These data packets are sent upwards to the top neighbor of each processor cell, eventually reaching the control processor. The control processor then reassembles the computed data, using the unique instruction IDs, into a meaningful computation result.

During the first cycle of shift-out mode, each processor cell sends a data packet to its top neighbor. In subsequent cycles, all processor cells except the cells on the bottommost row will sense that there is an incoming data packet, and therefore they will pass the data packet from their bottom neighbor to their top neighbor, rather than pass up one of their own data packets. In this way, the edge bus between the NanoBox Grid and the control processor will first receive one data packet from each processor cell, from the topmost processor cells to the bottommost processor cells. The edge bus between the NanoBox Grid and the control processor will receive all of the data packets from the bottommost processor cells, then from the second to bottommost processor cells, and so on until all of the processor cells have transferred the contents of their memories to data packets and shifted out those data packets.

The unique instruction ID allows the NanoBox Processor Grid to interleave the data packets. That is, the control processor uses this unique instruction ID to reassemble the computed data so that it does not need to receive the computed data in any particular order.

3.3 Architecture of the Individual Processor Cells

Each processor cell contains a simple ALU, a small amount of read/writable memory, and a communication router, as shown Figure 5. The logic, architecture and behavior of each block are described below.

The simple ALU has the four instructions shown in Table 1. The ALU accepts two 8-bit inputs and produces an 8-bit output, based on the value of the 3-bit opcode. The ALU is active only when the processor cell is in compute mode.

In this initial investigation, the memory unit of a processor cell contains 32 words. (The size of the memory is arbitrary and may be increased in future investigations.) This memory is active during all modes (shift-in, compute, and shift-out) of the processor cell.

Between the nbox-alu and the nbox-memory is the nbox-aluctrl, or ALU control unit. This logic block is active only when the processor cell is in compute mode.

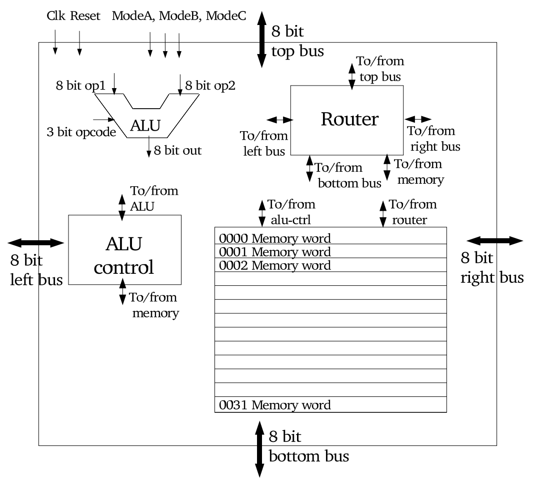

Figure 5: Diagram and detailed architecture of a single NanoBox processor cell.

Table 1: ALU Instruction Set.

Opcode	Instruction	Action
000	AND	Operand1 AND Operand2
001	OR	Operand1 OR Operand2
010	XOR	Operand1 XOR Operand2
111	ADD	Operand1 + Operand2

The nbox-aluctrl reads a word from the nbox-memory and computes the majority value of the three `data-valid` bits. If the memory word contains valid data, nbox-aluctrl computes the majority value of the three `to-be-computed` bits. If the memory word contains valid data which has yet to be computed, nbox-aluctrl sends the two operands and the opcode to nbox-alu. After the computation, the memory word is reassembled and sent back to the nbox-memory for storage.

The nbox-aluctrl continues looping through each word of the nbox-memory as long as the processor cell is in compute mode, as described in Section 3.2.2.

The nbox-router will serve to monitor the four 8-bit wide top, bottom, right and left processor cell buses during all modes of operation (shift-in, compute, and shift-out). If a packet is sensed on any of these buses, the router will examine the destination processor cell ID and determine what to do with the packet. One of five situations follows this ID examination: (1) Send Left if column address $>$ cell ID; (2) Send Right if column address $<$ cell ID; (3) Send Up if row address $>$ cell ID; (4) Send Down if row address $<$ cell ID; (5) Keep Here if destination ID $=$ cell ID.

The router also will translate incoming data packets to memory words during shift-in mode, and will translate memory words into outgoing data packets during shift-out mode.

4 Evaluation Methodology

We use detailed VHDL simulations of the NanoBox Processor Grid to evaluate the effectiveness of our recursive fault masking approach in the presence of random transient errors. For this investigation, we focus on the error coverage provided by different bit-level and module-level fault tolerance techniques. An analysis of the area, power, and timing overhead of our bit-level fault tolerance techniques was conducted in [16]. Module-level fault tolerance area, power, and timing overhead has been studied in great detail [31] and is therefore not addressed in this evaluation.

For our concept demonstration, the NanoBox Processor Grid is targeted at image processing applications. We model a single processor cell and test the cell with the computations needed to reverse the colors of a bitmap and to perform hue shifts of a bitmap. We chose these data parallel, streaming applications to begin our evaluations of the NanoBox Processor Grid due to the inherent parallelism and natural resilience to errors in this class of applications.

The unique instruction ID used in the memory words within a processor cell corresponds to a pixel ID. In this way, the control processor can break the image into processor cell-sized pieces and then reassemble the altered image after computation.

To allow detailed analysis in this investigation, we model only the ALU [25] of the processor cell. This focus allows us to evaluate three different techniques for bit-level fault tolerance and three different techniques for module-level fault tolerance. In all, we model nine different implementations of the NanoBox ALU, each with a different combination of bit-level and module-level fault tolerance techniques. Modeling system-level fault tolerance, i.e. entire processor cell failover and recovery, is outside the scope of this initial investigation and is left to future work. As a baseline for comparison, we also model a traditional CMOS ALU that incorporates no bit-level redundancy and does not use lookup tables for its logic. The naming scheme for our various ALUs is shown in Table 2. All of our ALUs are constructed using the VHDL language and simulated using the Synopsis VHDL software tools.

In our simulations, we inject errors in the NanoBox ALUs by XORing the lookup table bit strings with a fault mask [26], as shown in Figure 6(a). We inject faults in the CMOS ALUs again by XORing nodes between transistors with a fault mask, as shown in Figure 6(b). Wherever a 1 bit appears in this mask, it will cause the corresponding line to invert its state, thereby simulating a bit flip error. After each ALU computation, we generate a new fault mask, thereby modeling uniformly distributed random transient device faults. For this investigation, we do not model faults in the lookup table error detector or corrector. However, we do model module-level error detector and corrector faults by using a lookup table for the module voter. This module voter lookup table, as with the lookup tables within the ALU, has errors injected on its bit string. For the time redundancy ALUs, we also model bit flips in the stored inter-operation ALU results. The middle column in

Table 2 shows the total number of possible fault injection sites for each ALU.

Due to the high error rates we are evaluating, we ramp the error rate based on the percent of fault injection points in an ALU implementation, rather than injecting a fixed number of errors. In other words, for a given experiment, the space redundancy ALUs will have more injected errors than the CMOS ALUs. However, the ratio of injected errors to error-injection sites is held constant across the twelve ALU implementations in a given experiment.

The failure rates of contemporary CMOS devices are on the order of 50,000 FITs [2]. One raw FIT, or Failure in Time, is defined as one transistor malfunction or other device upset that results in a bit flip in 10^9 hours of operation. Consequently, this failure rate corresponds to 50,000 errors per 10^9 hours, or one error per 20,000 hours, which is approximately one error in two years. Molecular device technologies are not yet mature enough to have citable FIT rates. Consequently, we chose to model our ALUs with a wide distribution of fault percentages to exaggerate what can reasonably be expected when real devices become available.

During each computation, we force a given fraction of the fault injection points to flip their states thereby simulating faults. Note, however, that not all of the injected faults will necessarily produce observable errors in the output. For instance, if one input to an AND gate is at logic 0, its output is independent of the value on the other input. Consequently, injecting a fault on this other input will not produce an error in the output of the gate. We run simulations at eighteen different injected fault percentages: 0, 0.05, 0.1, 0.5, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 20, 30, 50, 75. We can translate these fault percentages into FIT rates for our NanoBox ALUs by assuming the ALU performs one calculation every 0.5 nanoseconds. This clock period, which equates to a 2 Ghz clock rate, was determined by device-level simulations in [16]. FIT rates are then determined by computing the ratio of the number of injected errors per 0.5 nanoseconds. For example, the *aluss* implementation has 5040 nodes on which faults could be injected. Injecting faults on 1 percent of these nodes would produce 50 total faults per 0.5 nanosecond clock. This fault rate then corresponds to $3.6 * 10^{14}$ errors per hour, or a FIT rate of $3.6 * 10^{23}$. The fault percentages, and the resulting FIT rates, used in these simulations equate to significantly higher raw FIT rates than the raw failure rates of current CMOS device technologies.

Our test workload bitmap contains 64, 8-bit pixels. Reversing the video of this bitmap is accomplished by computing the XOR of each pixel with a mask of ”11111111”. We shift the hue of the bitmap by adding a constant ”00001100” to each pixel. The reverse video and hue shift workloads each have 64, 8-bit computations. We simulate each ALU, running each workload, at each fault percentage, five times. Each computation of each workload simulation uses a different, randomly generated, fault mask.

Table 2: ALU Naming Conventions and the potential number of fault injection sites.

ALU Type	Potential Fault Points	Description
aluncmos	192	Traditional CMOS ALU with no module-level redundancy
alunh	672	NanoBox ALU with Hamming information code lookup tables and no module-level redundancy
alunn	512	NanoBox ALU with no code lookup tables and no module-level redundancy
aluns	1536	NanoBox ALU with triplicated bit string lookup tables and no module-level redundancy
aluscmos	657	Three copies (module-level space redundancy) of Traditional CMOS ALU
alush	2205	Three copies (module-level space redundancy) of NanoBox ALU with Hamming information code lookup tables
alusn	1680	Three copies (module-level space redundancy) of NanoBox ALU with no code lookup tables
aluss	5040	Three copies (module-level space redundancy) of NanoBox ALU with triplicated bit string lookup tables
alutcmos	684	One Traditional CMOS ALU, calculating three times (module-level time redundancy)
aluth	2232	One NanoBox ALU with Hamming information code lookup tables, calculating three times (module-level time redundancy)
alutn	1707	One NanoBox ALU with no code lookup tables, calculating three times (module-level time redundancy)
aluts	5067	One NanoBox ALU with triplicated bit string lookup tables, calculating three times (module-level time redundancy)

5 Simulation Results and Discussion

Figures 7, 8, and 9 present the percent of correct computation versus the injected fault rates for the nine NanoBox ALUs and three conventional CMOS ALUs. Figures 7, 8, and 9 group this data by module-level fault tolerance technique. Figure 7 shows a system in which the ALUs have only fine-grained bit-level redundancy, but no module-level redundancy. Figure 8 shows the results for ALUs using time redundancy, i.e. the ALU calculates the instruction three times, storing the three inter-operation results. Figure 9 shows ALUs that use space redundancy, i.e. three copies of the ALU, operating concurrently. For both the time and space redundancy ALUs, the overall result is the majority vote of the three individually computed results.

Our metric for comparison, *average percent correct computations*, is computed by taking the average of the percentage of correct computations produced by five trials of the two workloads for each ALU implementation, where each these ten total trials uses a different randomly generated fault injection mask. In other words, each data point in Figures 7, 8, and 9 is the average of ten data samples of a given ALU, at a given injected fault percentage. The standard deviation was less than 10.0 percentage points for all but six of the 216 plotted points. Of these six data points, the maximum standard deviation was 24.51. Given the spread between the data series in Figures 7, 8, and 9, the variance across the ten samples of each data point was statistically insignificant.

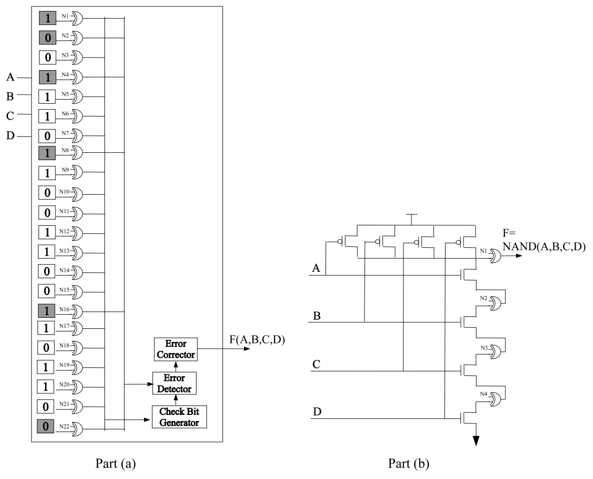

Figure 6: Part (a). Method for injecting faults into bit-level NanoBoxes. Lookup table bits are flipped via XOR gates. Part (b). Method for injecting faults into CMOS ALUs. Nodes between transistors are flipped via XOR gates. The fault mask vector N is regenerated each cycle with a randomly distributed number of faults.

Figure 7 shows that, in a system with no module level redundancy, the NanoBox ALU with the triplicated bit string lookup table (*aluns*) produced the best results, being able to maintain greater than 98 percent correct execution with injected fault rates as high as 2 percent. This configuration further maintained better than 60 percent correct computation with injected fault rates as high as 9 percent. All of the other ALUs, *alunn*, *alunh*, and *aluncmos*, dropped below 60 percent correct computation at injected error rates below 3 percent.

The *alunn* configuration, which is a NanoBox ALU with no redundancy of any form, was better than the ALU with Hamming information code (*alunh*) across all the fault injection percentages. This result is surprising since we expected the bit-level information redundancy to mask low numbers of injected faults. We attribute the poor results of the information-coded ALUs to false positives caused by errors in bits which are not addressed by the lookup table inputs. The information code error detector and error corrector access all of the stored bits in the lookup table bit string when determining the error syndrome. In the no-code and in the triple module redundancy lookup table, used in *alunn* and *aluns*, respectively, only the bits actually being addressed for a specific computation are accessed. Errors on non-addressed bits in the lookup table bit string are ignored in these configurations. These differences suggest that information codes are not a good choice for this type of bit-level fault tolerance.

The CMOS ALU (*aluncmos*) had the worst fault tolerance results, dropping to 39 percent correct computation at only 1 percent injected errors. The percent of correct computation for this ALU dropped to 9 percent at 3 percent injected errors and was at nearly 0 percent correct computation for all higher densities of injected errors.

Figure 8, with bit-level fault tolerance techniques for a system with module-level time redundancy, and Figure 9, with bit-level fault tolerance techniques for a system with module-level space redundancy, show nearly identical results as in Figure 7. Comparing the three different triplicated bit string lookup techniques across Figures 7, 8, and 9, that is, ALUs *aluns*, *aluts*, and *aluss*, we see very similar data points. We also see very little difference when comparing *alunn*, *alutn*, and *alusn*, comparing *alunh*, *aluth*, and *alush*, and comparing *aluncmos*, *alutcmos*, and *aluscmos*. These similarities across bit-level techniques indicate that, for device technologies with the extremely high fault rates simulated in these experiments, module-level fault tolerance becomes ineffective. By eliminating the module-level fault tolerance altogether, as with *aluns*, we obtain nearly identical results as with module-level space redundancy, as in *aluss*.

Overall, our best results were obtained by combining space redundancy at the bit-level and space redundancy at the module-level. With this configuration, *aluss*, we obtain 98 percent (or better) correct computation at injected error rates as high at 3 percent. With a workload of 64 pixels, a 98 percent correct computation means that fewer than one pixel had an incorrect value. In contrast, the CMOS ALU with no module-level redundancy, *aluncmos*, had only 9 percent correct computation with 3 percent injected errors. This means that only 6 out of the 64 pixels had the correct value. The FIT rate for the *aluss* ALU, at 3 percent injected errors, is 10^{24}, which is tremendously greater than FIT rates of conventional CMOS devices.

These results show that triple modular redundancy (i.e. space redundancy) at the bit-level, combined with triple modular redundancy at the module level can produce 98 percent correct computation, despite FIT rates twenty orders of magnitude higher than modern-day CMOS FIT rates. By triplicating at the bit-level and triplicating again at the module-level, we incur area overhead on the order of 9x. With the expected small size and unprecedented level of integration available with non-silicon, nanoscale device technologies, this 9x overhead may be quite reasonable given the very high fault tolerating capabilities obtained with this moderate overhead.

6 Related Work

Since molecular nanodevices are not yet mature enough to fabricate complex designs, the field of nanocomputing is relatively unexplored. Consequently, there is not a great deal of prior work directly related to our NanoBox Processor Grid project. Instead, the relevant prior work falls into three distinct categories: classic fault-detection techniques, external reconfiguration, and processor-in-memory architectures. The NanoBox Processor Grid project aims to learn from these distinct fields and to bridge the knowledge gaps among them.

6.1 Classical Fault Detection

A *watchdog processor* is a general technique for detecting faults in which an auxiliary processor runs concurrently with the main processor [20]. The watchdog processor executes the same program as the main processor using the same inputs. A *checker* periodically compares the outputs of the two processors and signals a fault

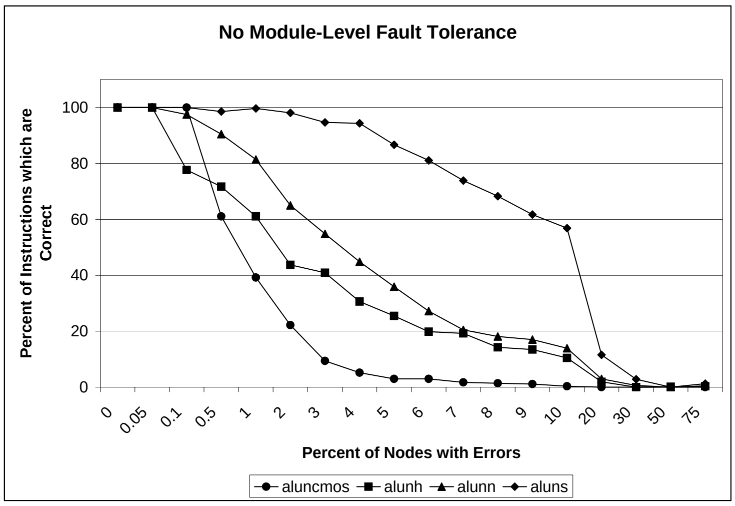

Figure 7: Percent correct instructions versus injected error rates, for ALUs with no module-level fault tolerance.

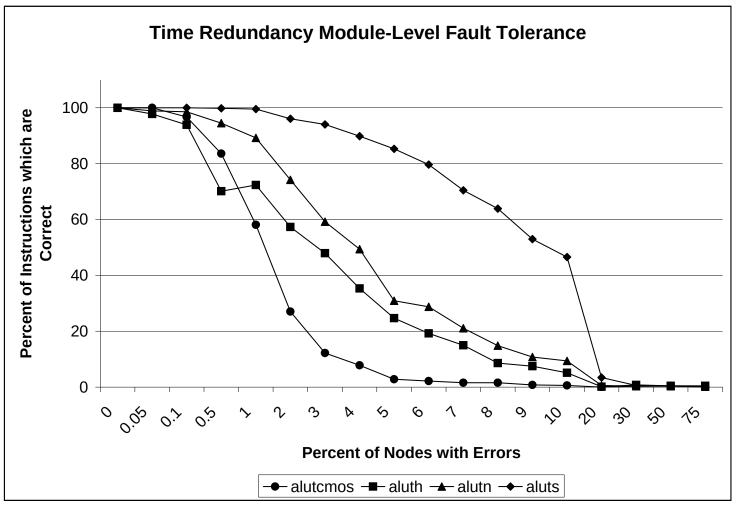

Figure 8: Percent correct instructions versus injected error rates, for ALUs with module-level time redundancy.

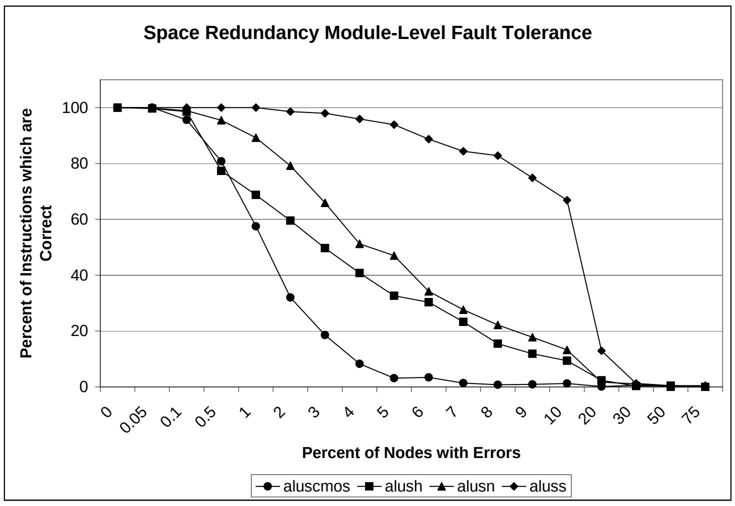

Figure 9: Percent correct instructions versus injected error rates, for ALUs with module-level space redundancy.

if they fail to match. The Simultaneous and Redundantly Threaded (SRT) processor [27], the AR-SMT processor [29], and DIVA [1] are examples of the watchdog processor approach. Examples of existing computer systems that use a complete replicate-and-compare approach to fault detection are the Compaq NonStop Himalaya system [31], which runs two identical off-the-shelf microprocessors in lockstep, and the IBM S/390 G5 [32], which replicates pipeline stages on the chip. Triple modular redundancy has been proposed for detecting and correcting faults in an FPGA system [9]. A variety of other circuit techniques have been proposed [10, 14, 23, 28, 33] that use both space and time redundancy for fault-tolerance.

Since nanoscale devices have limited drive capability and, therefore, only nearest neighbor communication, sending validation signals to a separate chip, via a bus or via a wire of back-to-back molecular nanodevices, will incur an extraordinary latency. To address this problem, our approach integrates fault-tolerance into the chip at a much finer granularity than a watchdog processor.

6.2 External Reconfiguration

External fault-tolerance and device reconfiguration has been proposed as a possible way to cope with unstable molecular devices. In the Teramac [13] project, a digital system was built out of unreliable field-programmable gate arrays (FPGAs). An external testing computer was connected to the Teramac to periodically survey the blocks of FPGAs and identify which blocks had incurred errors since the last fault survey. Faulty blocks were disabled and the connections to and from those faulty blocks were rerouted to neighboring blocks. Although Teramac was built out of FPGAs, not nanodevices, the unreliable FPGAs had characteristics similar to the types of molecular nanodevices currently being proposed by other researchers. The actual molecular devices which may be used in the Teramac are still under development.

The Phoenix [12] project proposed using external CMOS circuitry to periodically survey the computer system for faulty logic blocks. As in the Teramac project, once a faulty block is identified, the connections to and from that block are removed and/or rerouted to functioning blocks.

Periodic system testing becomes a critical bottleneck as computer systems scale in size. If blocks are laid out in a two-dimensional grid, every additional block adds several more connection points to the system with one or more connections into the block and one or more connections out of the block. This super-linear increase in connection points with additional blocks leads to an exponential increase in system checking time as more blocks are added to the system.

Our NanoBox architecture addresses the system check bottleneck by distributing the checking circuitry into the logic blocks themselves. In this way, system checks can be performed simultaneously with the actual computation inside the NanoBox wall. Errors are identified and corrected on the fly, rather than during a periodic system-wide validation survey. Reconfiguration is incorporated into the NanoBox Processor Grid by the heartbeat monitoring and deploying entire processor cell failover and recovery.

6.3 Processor in Memory

Various architectures have been proposed which explore merging processing and memory storage into a grid of computational cells [4, 19, 21, 22, 24]. By bringing computational logic closer to the data storage, these architectures aim to speed computation and reduce power dissipation. None of these architectures has explicitly been designed to address device defects and soft errors, however.

A subclass of processor-in-memory architectures, so-called *computation cache* architectures [34], take the philosophy that computation, or instructions, should stay in place and data should move through the chip. This counters traditional superscalar architectures where data stays resident in registers while the instruction, or opcode, changes.

Our work uses a processor-in-memory architecture due to the expected inherent lack of long distance signaling available in future technologies. Our approach integrates the most applicable features of each of the prior processor-in-memory architectures and adapts these features to the characteristics of nanotechnology devices. Also, the distributed redundancy inherent to a processor-in-memory architecture provides an excellent foundation for this hierarchical approach to fault encapsulation.

7 Future Work

At present, we have a behavioral VHDL model of the entire NanoBox Processor cell, and a lookup table VHDL model of the NanoBox Processor cell ALU. Our foremost future work is to convert the entire processor cell, including the router and alu-control modules, into lookup tables. In this way, we can expand our fault injection experiments and analyze the effect of high fault rates on control logic.

Another topic of future work is to model the entire NanoBox Processor Grid so as to evaluate techniques for processor cell failover and recovery. Of key interest are the protocols for how much heartbeat monitoring is needed by the watchdog and how the control microprocessor should reroute data assigned to a failed processor cell.

We also plan to develop a cycle-based, full-system simulator for running a range of application-level workloads. In this way, we can evaluate how the NanoBox Processor Grid may be adapted for non-streaming workloads.

8 Conclusions

In this paper, we introduce the Recursive NanoBox Processor Grid, a fault-tolerant processor to be fabricated with nanotechnology devices. We use VHDL to model and simulate one processor cell of this grid. The processor cell contains an ALU, a memory, and a communication unit.

The fundamental logic unit of the NanoBox Processor Grid is a gate array-style lookup table which uses error correction on the function truth table. This error correction is able to correct single bit flips within the lookup table, thereby presenting a robust logic block to higher levels of the processor cell design.

Our analysis shows motivating results for using fine-grained, bit-level fault tolerance techniques in computing systems fabricated from non-silicon, nanoscale device technologies. By using a recursive space redundancy fault tolerance approach at both the bit and module levels, we can achieve 100 percent correct computation for our example computational workloads despite FIT rates above 10^{23}, which is twenty orders of magnitude higher than contemporary CMOS device technologies. We conclude that the recursive NanoBox processor grid is a viable approach for dealing with the high transient error rates and large numbers of manufacturing defects expected with future nanotechnology devices.

9 Acknowledgements

The authors would like to thank the University of Minnesota NanoBox Research group for their input during discussions related to this work: 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, and Gary Carpenter for lively discussions regarding the implementation and niche of the NanoBox Processor Grid. 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] Todd Austin. DIVA: A reliable substrate for deep submicron microarchitecture design. In *International Symposium on Microarchitecture*, pages 196–207, 1999.

[2] Robert Baumann. The impact of technology scaling on soft error rate performance and limits to the efficacy of error correction. In *International Electron Devices Meeting (IEDM)*, pages 329–332, 2002.

[3] George Bourianoff. The future of nanocomputing. *IEEE Computer*, pages 44–53, August 2003.

[4] Jurgen Buddefeld and Karl E. Grosspietsch. Intelligent-memory architecture for artificial neural networks. *IEEE Micro*, May-June 2002.

[5] Cristian Constantinescu. Trends and challenges in VLSI circuit reliability. *IEEE Micro*, pages 14–19, July-August 2003.

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

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

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

[9] S. D'Angelo, C. Metra, S. Pastore, A. Pogutz, and G.R. Sechi. Fault-tolerant voting mechanism and recovery scheme for TMR FPGA-based systems. In *IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, November 1998.

[10] Manoj Franklin. Incorporating fault tolerance in superscalar processors. In *Proceedings of High Performance Computing*, 1996.

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

[12] Seth Copen Goldstein and Mihai Budiu. Nanofabrics: Spatial computing using molecular electronics. In *International Symposium on Computer Architecture (ISCA)*, July 2001.

[13] James R. Heath, Philip J. Kuekes, Gregory S. Snider, and R. Stanley Williams. A defect-tolerant computer architecture: Opportunities for nanotechnology. *Science*, 280:1716–1721, June 1998.

[14] John G. Holm and Prithviraj Banerjee. Low cost concurrent error detection in a VLIW architecture using replicated instructions. In *International Conference on Parallel Processing*, 1992.

[15] Hiroki Iwamura, Masamichi Akazawa, and Yoshihito Amemiya. Single-electron majority logic circuits. *IEICE Transactions on Electronics*, E81-C(1):42–48, 1998.

[16] AJ KleinOsowski and David J. Lilja. The nanobox project: Exploring fabrics of self-correcting logic blocks for high defect rate molecular device technologies. In *IEEE Symposium on VLSI (ISVLSI)*, February 2004.

[17] AJ KleinOsowski, Priyadarshini Ranganath, Mahesh Subramony, Vijay Rangarajan, Kevin KleinOsowski, and David J. Lilja. The nanobox: A self-correcting logic block for emerging process technologies with high defect rates. Technical Report ARCTiC 03-02, University of Minnesota, Department of Electrical and Computer Engineering, ARCTiC Laboratory, June 2003.

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

[19] Guangming Lu, Hartej Singh, Ming-Hau Lee, Nader Bagherzadeh, Fadi Kurdahi, and Eliseu M. C. Filho. The morphosys parallel reconfigurable system. In *Euro-Par*, 1999.

[20] A. Mahmood and E. J. McCluskey. Concurrent error detection using watchdog processors – A survey. In *IEEE Transactions on Computers*, volume 37 (2), pages 160–174, February 1998.

[21] Ramadass Nagarajan, Karthikeyan Sankaralingam, Doug Burger, and Stephen W. Keckler. A design space evaluation of grid processor architectures. In *International Symposium on Microarchitecture (MICRO)*, 2001.

[22] Mark Oskin, Justin Hensley, Diana Keen, Frederic T. Chong, Matthew Farrens, and Aneet Chopra. Exploiting ILP in page-based intelligent memory. In *International Symposium on Microarchitecture (MICRO)*, 1999.

[23] Janak H. Patel and Leona Y. Fung. Concurrent error detection in ALUs by recomputing with shifted operands. In *IEEE Transactions on Computers*, volume 31 (7), pages 589–595, July 1982.

[24] David Patterson, Thomas Anderson, Neal Cardwell, Richard Fromm, Kimberly Keeton, Christoforos Kozyrakis, Randi Thomas, and Katherine Yelick. A case for intelligent RAM: IRAM. *IEEE Micro*, April 1997.

[25] Priyadarshini Ranganath. Recurvise NanoBoxes: Reliable Circuits and Computer Architectures for Nanotechnology Devices. Masters of Computer Engineering Plan B Project, University of Minnesota, October 2003.

[26] Vijay Rangarajan. The NanoBox ALU: Design, Implementation, and Fault Simulation. Masters of Computer Engineering Plan B Project, University of Minnesota, October 2003.

[27] Steven Reinhardt and Shubhendu Mukherjee. Transient fault detection via simultaneous multithreading. In *International Symposium on Computer Architecture*, pages 25–36, June 2000.

[28] Dennis A. Reynolds and Gernot Metze. Fault detection capabilities of alternating logic. In *IEEE Transactions on Computers*, volume 27 (12), pages 1093–1098, December 1978.

[29] Eric Rotenberg. AR-SMT: A microarchitectural approach to fault tolerance in microprocessors. In *International Symposium on Fault-Tolerant Computing*, 1999.

[30] Premkishore Shivakumar, Michael Kistler, Stephen W. Keckler, Doug Burger, and Lorenzo 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.

[31] Daniel P. Siewiorek and Robert Swarz. *Reliable Computer Systems: Design and Evaluation*. A. K. Peters, third edition, 2001.

[32] T. J. Slegel and et al. IBM's S/390 G5 microprocessor design. In *IEEE Micro*, pages 12–23, March/April 1999.

[33] G. S. Sohi, M. Franklin, and K. K. Saluja. A study of time-redundant fault tolerance techniques for high-performance pipelined computers. In *International Symposium on Fault-Tolerant Computing*, pages 436–443, 1989.

[34] Steven Swanson and Mark Oskin. Towards a universal building block of molecular and silicon computation. In *Workshop for Non-Silicon Computation, International Symposium for High-Performance Computer Architecture (HPCA)*, February 2002.

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