Nostradamus: Low-Cost Hardware-Only Error Detection for Processor Cores

Ralph Nathan and Daniel J. Sorin
Department of Electrical and Computer Engineering
Duke University

Abstract - **We propose a new, low-cost, hardware-only scheme to detect errors in superscalar, out-of-order processor cores. For each instruction decoded, Nostradamus compares what the instruction is expected to do against what the instruction actually does. We implement Nostradamus in RTL on top of a baseline superscalar, out-of-order core, and we experimentally evaluate its ability to detect injected errors. We also evaluate Nostradamus's area and power overheads.**

I. INTRODUCTION

In this paper, we propose a new, low-cost scheme for detecting errors in superscalar, out-of-order processor cores. Our scheme, called Nostradamus, operates on a simple principle: for each instruction decoded, Nostradamus compares what the instruction is expected to do against what the instruction actually does. Instructions can modify architectural registers, memory, and the program counter (PC); Nostradamus perfectly forecasts (which is origin of its name) during the Decode stage what each instruction (except for conditional branches) should do to architectural state. As the instruction proceeds through the pipeline, Nostradamus records what the instruction does. When the instruction commits, Nostradamus compares what it was supposed to do against what it actually did. We present the design of Nostradamus in Section II.

As we explain in Section III, we have implemented Nostradamus in Verilog on top of a baseline superscalar core distributed by the FabScalar group [4]. FabScalar, which is designed in RTL, enables us to experimentally evaluate Nostradamus's ability to detect errors, as well as its area and power overheads. Our results for area and power (Section III) show that Nostradamus's overheads are modest, less than 11% for area and less than 6% for power. Our results for error detection (Section IV) show that Nostradamus successfully prevents the vast majority of injected errors from causing silent data corruptions (SDCs).

Because core error detection is an important problem, there have been many different schemes proposed prior to Nostradamus. We summarize prior work in Table I and note that each scheme has at least one significant drawback. Our goal for Nostradamus is to provide comprehensive error detection for the core without any of these major drawbacks.

In this work, we make three contributions:

- We propose a novel, low-cost, all-hardware error detection scheme for superscalar cores.
- We develop an RTL implementation of Nostradamus.
- We experimentally evaluate the RTL implementation.

II. NOSTRADAMUS DESIGN

The key idea behind Nostradamus is to compare what the core does to what it is expected to do. For each instruction, Nostradamus compares the instruction's expected impact on architectural state to the actual impact the instruction's execution has on architectural state. Any difference between the two reveals an error. This approach is similar in philosophy to prior work [7][2][8] but with advantages listed in Table 1. Although Nostradamus can detect errors in simple, in-order cores, we focus on superscalar, out-of-order cores.

A. Overview

Nostradamus operates on a per-instruction basis. After an instruction is fetched, it is decoded. In the Decode stage, the core has all of the information required to determine what impact the instruction will have on the core's *architectural* state. This state includes the architectural registers, memory, and program counter (PC). This architectural impact is independent of the microarchitecture of the core, i.e., Nostradamus is agnostic as to *how* the core will execute the instruction. Nostradamus cares only about the "bottom line" of how the instruction modifies architectural state. Checking at the architectural level, rather than the microarchitectural level, enables Nostradamus to comprehensively detect errors throughout the core. For example, by checking updates to architectural registers, rather than physical registers, Nostradamus can detect errors in register renaming.

In Figure 1, we illustrate Nostradamus. When a fetched instruction is decoded, Nostradamus's SetExpectation unit determines what the instruction should do, i.e., its *expectation*. In the out-of-order execution engine—everything between Decode and Commit—Nostradamus tracks the instruction's *history* (i.e., its modifications to architectural state). Just prior to committing the instruction, Nostradamus's CheckExpectation unit compares the instruction's history to its expectation. A mismatch indicates an error.

As described thus far, Nostradamus comprehensively detects errors in the core's dataflow and control flow decision making processes. For example, if an error causes the core to read from the wrong register, the history will not match the expectation. Similarly, if an error causes the core to

978-3-9815370-2-4/DATE14/©2014 EDAA

TABLE I. QUALITATIVE COMPARISON OF STATE-OF-THE-ART CORE ERROR DETECTION SCHEMES

core error detection scheme	error detection coverage	major drawback
Nostradamus (this paper)	vast majority[a] of transients and permanents	
DIVA [1]	virtually all[b] transients and permanents	large area and power overheads to check small cores; adds many new datapaths to core
redundant multithreading [15][13]	vast majority[a] of transients	large energy and performance overheads
register dataflow checking [2][8]	no RTL designs evaluated, but expect vast majority[a] of transients and permanents in dataflow	no coverage of control flow errors; may require recompilation (i.e., access to source code) [8]
Argus [7]	vast majority[a] of transients and permanents	requires recompilation (i.e., access to source code)
periodic built-in self-test (e.g., BulletProof [17])	virtually all[b] permanents	no coverage of transient errors
software-level anomaly detection (e.g., SWAT [6])	vast majority[a] of permanents	unclear detection of transients; unbounded error detection latency
software redundancy (e.g., SWIFT [14])	no RTL designs evaluated, but expect majority of transients	very large energy and performance overheads; lower error coverage than hardware schemes
Sampling + DMR [10]	virtually all[b] permanents	no coverage of transient errors

[a] We use "vast majority" detected to denote that less than 10% of errors lead to silent data corruptions.

[b] We use "virtually all" detected to denote that less than 1% of errors lead to silent data corruptions.

incorrectly advance the PC, the history will not match the expectation.

To be complete, Nostradamus must also detect errors in values that are computed in functional units and maintained in storage structures. Detecting errors in computations and storage is straightforward, and we adopt well-known solutions for both. Nostradamus detects errors in computations with residue coding [16] and errors in storage with error detecting codes (e.g., parity).

B. Microarchitectural Design

We now describe how we integrate Nostradamus into the design of a superscalar, out-of-order core. When designing Nostradamus, our main goals were to avoid adding new datapaths and to avoid modifications to complicated, latency-critical units like register renaming.

Instruction Fetch

Nostradamus has no impact whatsoever on the Fetch stage. Nostradamus does not detect errors in Fetch, other than detecting errors in updating the program counter (discussed later in this section).

Decode and SetExpectation Unit

In the Decode stage, Nostradamus adds a SetExpectation unit that operates *in parallel* with the normal instruction decode logic. The normal instruction decode logic processes an instruction to produce the signals to control the pipeline accordingly. The SetExpectation unit has the analogous but simpler task of determining its expectation, i.e., how the instruction will modify the core's *architectural* state.

The SetExpectation unit processes the instruction to determine the instruction's expectation, which is a function of the instruction's operation type, architectural register inputs, immediate input operands, architectural register outputs, and next PC. For example, the MIPS instruction "`add $r3, $r1, $r7`" would expect to: perform an addition; use input operands $r1 and $r7; write to $r3; and update PC to PC+4.

Some instructions perform multiple operations. An example of this is the MIPS instruction "`load $r2, 4[$r1]`". This load instruction is expected to compute an address from an immediate and a register value ($r1+4) and then perform a load at that address. Nostradamus checks that both the ALU and the Load Store Unit are accessed and that the correct operations occur within each of them.

Signatures: Logically, Nostradamus uses all of the information in an instruction's expectation, but the cost of maintaining all of this information would be impractical. Instead, the SetExpectation unit hashes this information to create an *S*-bit value we call the *Instruction Expectation Signature (IES)*. The choice of *S* is a design decision that enables the architect to trade off error coverage versus cost; as the value of *S* is increased, the error coverage improves (for reasons explained in Section II.D), but the cost of computing and maintaining the IES increases. For clarity of explanation, we assume for now that the IES contains all of the expectation information, and we re-visit how to incorporate hashed signatures later.

For implementation reasons explained later, we separate the IES for the PC, denoted IES(PC), from the IES for the registers and memory, denoted IES(RegMem). The IES(PC) is the expected value of the next PC, i.e., how the instruction is expected to affect the PC.

Control-Flow Instructions: For non-control-flow instructions (e.g., add), the expected next PC, IES(PC), is its PC plus the

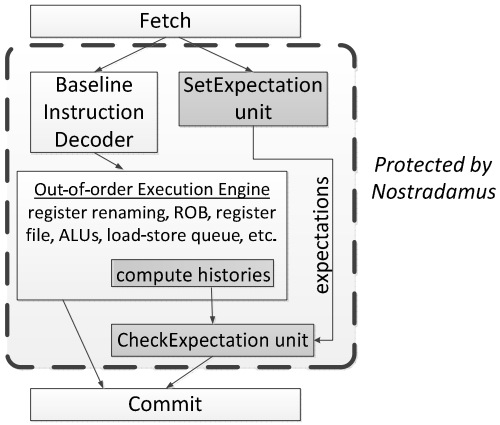

Fig. 1. High-level view of Nostradamus. Shaded units added for Nostradamus.

size of the instruction. For a control-flow instruction, the core does not know *a priori* what the next PC is going to be. However, the core predicts the next PC and Nostradamus uses this predicted next PC as the instruction's IES(PC). If the prediction is found to be incorrect in Execute, it will be updated then (as we explain in the section on Execute).

Micro-Ops: The FabScalar core, like many x86 cores, breaks some instructions into micro-ops. Nostradamus operates at the micro-op granularity because micro-ops effectively behave as instructions, i.e., read/write registers, modify the PC, etc. Nostradamus computes an expectation (IES) per micro-op and checks it at Commit.

Out-of-Order Execution Engine

In the out-of-order execution engine between Decode and Commit, Nostradamus must maintain each instruction's IES and compute each instruction's history. An instruction's history is computed with the same algorithm as the expectation, and the history is similarly hashed into an Instruction History Signature (IHS). As with the IES, the IHS is also separated into IHS(PC) and IHS(RegMem). An instruction's IHS(PC) is its PC.

We now discuss how Nostradamus is integrated into the out-of-order execution engine.

Register File and Register Renaming

The baseline core has a physical register file and an explicit renaming table. For purposes of computing the IHS, Nostradamus widens each register in the physical register file to include its current architectural register number. When an instruction writes to the register file it also writes the architectural destination register number along with the register. When an instruction reads from the register file, it also reads the architectural register number associated with the physical register. The architectural input register numbers then travel along with the instruction through the instruction queue, reorder buffer, etc., and are used to compute the IHS.

Nostradamus has no impact on register renaming, but it does detect errors in register renaming.

In-Flight Instruction State

The superscalar core manages the out-of-order execution of instructions with the register file (discussed already), instruction queue (IQ), reorder buffer (ROB), and load-store queue (LSQ). Nostradamus simply extends the IQ, ROB, and LSQ to hold signatures, as described in Table II.

Execute Stage and Functional Units

The functional units represent the primary place in the core where new architectural values are computed. Nostradamus enhances the Execute stage in three ways.

First, Nostradamus checks that computations of new values are correct. Detecting errors in functional units is a well-understood problem with well-tested solutions [16]. We adopt residue checking (also known as modulo checking). Our baseline processor has an integer adder and multiplier, and Nostradamus detects errors in both with residue checking with a modulus of 31. A larger modulus improves error detection coverage but increases area and power costs.

Second, Nostradamus must be aware of what operation is performed so that it can update the instruction's IHS accordingly. Thus, when the functional unit sends its result (with parity) to the register file, it sends the type of operation it performed (e.g., add, shift) to the ROB. For example, for an add instruction, the IHS is updated at the Execute stage in parallel with the computation. For a memory operation, the IHS is updated both at the Execute stage (where the address computation occurs) and at the Load Store Unit (where the memory operation occurs).

Third, Nostradamus may need to update the expectation for the next PC at Execute, because this is when the core resolves branch outcomes. If the branch's resolved target differs from the branch's predicted next PC, then Nostradamus changes the expected next PC, IES(PC), to the branch's resolved target. (Nostradamus detects errors in computing the resolved target.) Errors in branch prediction logic are not problematic because the worst case scenario is just a mis-prediction.

CheckExpectation Unit

Nostradamus detects errors with its CheckExpectation unit. Using the information that Nostradamus adds to the ROB, the CheckExpectation unit compares the IHS of each committing instruction to the instruction's IES (which is also in the instruction's ROB entry).

Checking updates of register and memory state: Checking an instruction's update of architectural register and memory state is straightforward. All of the instruction's history and expectation information are available in the ROB and can be compared. The only subtlety is that the error that is detected could be an error in a previous instruction. If a previous instruction wrote to the wrong register, then the error will not be detected until a later instruction tries to read that register.

Checking updates of the PC: Checking an instruction's update of the PC is somewhat more complicated because it involves pairs of instructions (rather than one instruction at a time). Assume that the most recently committed instruction is instruction *I* and the CheckExpectation unit is now checking the next instruction that is ready to commit, *I*+1. The CheckExpectation unit compares the PC of *I*+1 (which is part of *I*+1's history) to the expected next PC of instruction *I* (which is part of *I*'s expectation). Thus, for each instruction that commits, Nostradamus uses that instruction's IHS(PC) as well as that instruction's IES(PC).[1] In Figure 2, we illustrate an example of Nostradamus using the IES(PC) and IHS(PC) values to compare *I*+1's PC to *I*'s expected next PC. In the example, the core can commit 4 instructions per cycle and thus the CheckExpectation unit performs up to 4 comparisons per cycle. Nostradamus requires an extra register (the shaded "PC Check Reg" in the figure) to compare the IHS(PC) of the first instruction to commit in a cycle to the IES(PC) of the last instruction to commit in the previous cycle.

[1] Recall that IHS(PC) is the PC of the current instruction and IES(PC) is the expected PC of the next instruction to commit in program order.

Incorporating Lossy Signatures

For clarity, we have thus far assumed that the IES and IHS are complete, but we mentioned that we actually split each signature into two parts and hash the expectations and histories into lossy signatures.

We split the signature for the PC from the signature for the other architectural state due to implementation issues. Specifically, the IES sometimes needs to be updated in the Execute state for control flow instructions (i.e., if the core determines that a branch was mispredicted). If the IES were a single signature, updating it at this stage would be complicated. By splitting off the IES(PC), we greatly simplify this update.

We hash the signatures to reduce the cost of implementing Nostradamus. For all signatures, our implementation of Nostradamus uses CRC-5 as the hashing function. Thus each IES is a 10-bit quantity consisting of two CRC-5 values, one for IES(PC) and one for IES(RegMem). Similarly, each IHS is a 10-bit quantity consisting of two CRC-5 values for IHS(PC) and IHC(RegMem).

Protecting Architectural Values

As explained in Section II.A, architectural values produced by the core must be protected. These values live in the register file and LSQ (addresses of loads and stores and values to be written by stores). Architectural values also travel along the datapaths, including pipeline bypass paths. Nostradamus protects these values—in storage and datapath—using parity.

Nostradamus uses parity in a typical fashion except in the LSQ. In the LSQ, Nostradamus computes parity over the XOR of the address and the data. This use of parity, similar to a use in Argus [7], enables Nostradamus to detect when a load erroneously obtains a value from a store to the wrong address. Simply protecting the data with parity would only reveal errors in the data and would not detect accesses to incorrect addresses.

Nostradamus does not need to protect any other values because errors in them will be detected elsewhere. Such values include, for example, the IES value in the ROB; if this IES is corrupted by an error, the CheckExpectation unit will notice that this erroneous IES does not match the instruction's IHS. (This error is a "false positive"—Nostradamus has detected an error that did not exist in the baseline core but instead is in Nostradamus's own hardware.)

TABLE II. NOSTRADAMUS'S HARDWARE OVERHEADS

Core Modification	Cost (in bits), with 10-bit IES and IHS
structure modifications	
Register File: add arch register number [6 bits] and parity [1 bit] to each physical register	extra 7 bits per 32-bit register → 21.9 %
IQ: add arch destination register [6 bits] and IES(PC) [5 bits] to each IQ entry	extra 11 bits per 130-bit IQ entry → 8.5%
ROB: add IES [10 bits] and IHS(RegMem) [5 bits] to each ROB entry	extra 15 bits per 89-bit ROB entry → 16.9%
LSQ: add parity of XOR of data and address [1 bit] to each LSQ entry	extra 1 bit per 32-bit entry → 3.1%
SetExpection and CheckExpectation Units	a small amount of combinational logic
datapath modifications	
datapath carrying register value: add arch register number [6 bits] and parity [1 bit]	extra 7 bits per 32-bit register value → 21.9%
datapath carrying result of functional unit: add arch destination register [6 bits] and parity [1 bit]	extra 7 bits per 32-bit value → 21.9%

Watchdog Timer

Some errors do not cause incorrect behavior but rather cause the core to hang. For example, consider an error in the ROB that prohibits the oldest instruction from committing. We detect these errors with the well-known technique of a watchdog timer that considers an error to have occurred if no instruction has committed in the past ten thousand cycles.

C. *Summary of Hardware Overheads*

As discussed in Section II.B, our design of Nostradamus augments the baseline core by widening some structures and datapaths. We aggregate all of these previously described structure modifications in Table II. When computing percentage overheads, we assume that the IES and IHS are 10-bit quantities, as is the case in our implementation. The results show that Nostradamus's costs are relatively small.

D. *Potential Holes in Error Coverage*

As illustrated in Figure 1, Nostradamus is designed to detect errors between Fetch and Commit. Due to the way in which we have designed Nostradamus to be low-cost, there is some probability of Nostradamus failing to detect an error. We

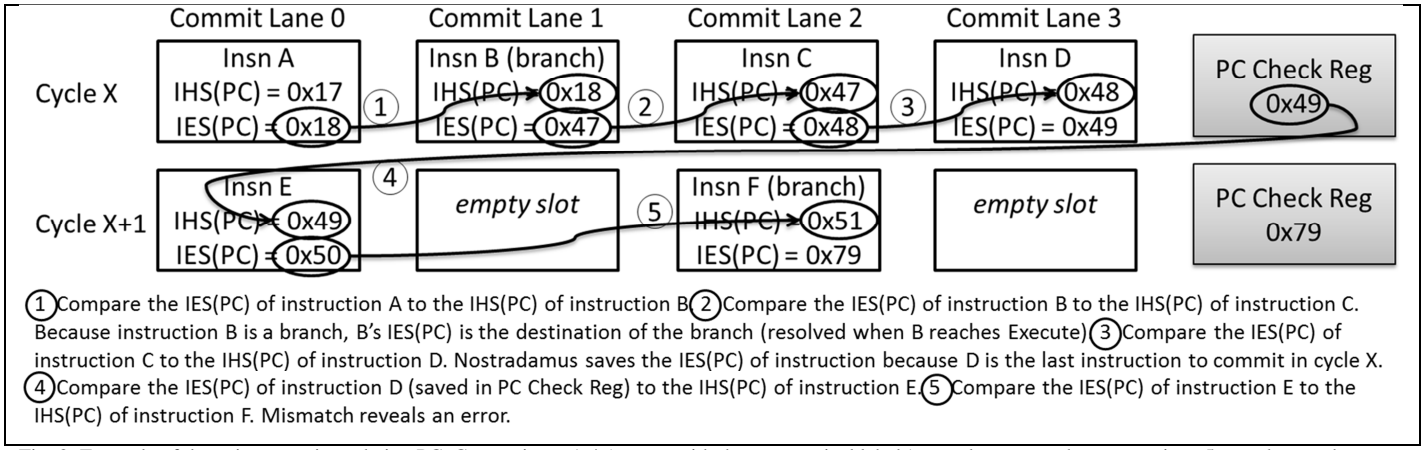

Fig. 2. Example of detecting error in updating PC. Comparisons 1-4 (arrows with those numerical labels) reveal no errors, but comparison 5 reveals error because of mismatch between Instruction E's IES(PC) and Instruction F's IHS(PC). Instruction F erroneously did not execute instruction it was expected to execute.

experimentally quantify this probability in Section IV, but here we provide the intuition for how errors can go undetected.

Our design of Nostradamus relies upon signatures in several places, and a signature is simply a lossy hash of a piece of information. Because of the lossy nature of a signature, there is a non-zero probability of aliasing, i.e., an error leading to a signature that just so happens to equal the signature of the error-free execution. Our choice of CRC-5 leads to 5-bit signatures (i.e., 5 bits for PC and 5 bits for RegMem) and thus a 2^{-5} probability of aliasing. Our experiments (not shown) suggest that CRC-5 is at or near the "sweet spot" in the trade-off between cost and the probability of aliasing.

Nostradamus's use of residue codes for checking functional units is also effectively a signature scheme, where the residue is a signature or lossy hash of a complete value. Similarly, residue coding is susceptible to a non-zero probability of aliasing that is a function of the modulus.

Nostradamus does not protect the Fetch stage of the core. As we see later, many of the injected errors that cause SDCs are errors in Fetch hardware.

Although Nostradamus completely covers errors in register dataflow, it does not completely cover errors in memory dataflow. Specifically, Nostradamus does not detect when an error causes a load of address *B* to obtain the value of the wrong store to address *B*. For example, Nostradamus will not detect if there are multiple stores to *B* in the LSQ and an error causes a load of *B* to fail to obtain the most recent store prior to the load [3]. Nostradamus does, however, detect when an error causes a load to obtain a value from the wrong address.

E. *No False Positives in Baseline Core*

Although Nostradamus can have "false negatives" (failing to detect an error), it is important to note that Nostradamus has zero "false positives" *in the baseline core*—that is, Nostradamus never signals an error if none has occurred in the baseline core hardware. Nostradamus can incur a false positive only when an error impacts Nostradamus's hardware (e.g., when an error corrupts an IES).

III. HARDWARE IMPLEMENTATION

We implemented Nostradamus in RTL, written in Verilog, on top of a baseline core distributed by the FabScalar group [4]. The core is a modestly out-of-order superscalar core with

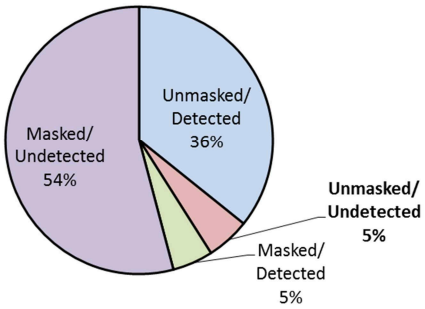

Fig. 3. Permanent errors

TABLE III. SUPERSCALAR CORE AND NOSTRADAMUS ADDITIONS

Parameter	value
Baseline core	
pipeline	depth: 13 stages, width: 4
register file	96 physical registers
out-of-order state	128-entry ROB, 64-entry load-store queue
caches	simulated functionally by FabScalar
Nostradamus additions	
residue checking modulus	31
size of IES/IHS	10 bits (5 bits for PC, 5 bits for RegMem)

the parameters and features described in Table III. Nostradamus requires no additional structures or paths but rather just slight widening of existing structures (e.g., register file and ROB) and paths. Nostradamus's hardware is entirely off the critical path and has no impact on clock period.

We used Synopsys CAD tools to floorplan and layout the core—both with and without Nostradamus—with 45nm standard cell technology from Nangate [9].

Area: We compare Nostradamus's area, relative to the baseline core. The results show that Nostradamus's total logic-only area overhead, without any storage structures, is 10.9%.

Power: Nostradamus consumes 34.1 mW of power, whereas the baseline consumes 32.2 mW, a difference of 5.6%. Delving a bit more deeply, Nostradamus uses 5.3% more dynamic power and 14.1% more static power.

Performance: Our implementation of Nostradamus has no impact on the core's performance. The checking of an instruction by the CheckExpectation unit is off the critical path and occurs in parallel with Commit. Even if the CheckExpectation unit's operation was on the critical path, Nostradamus could latch the data it needs and detect the error one cycle later.

IV. EXPERIMENTAL EVALUATION OF ERROR DETECTION

A. *Error Injection Methodology*

Transient error injection is a well-known challenge [5][12] because of the scale of the problem. For a design with *W* wires that runs a benchmark for *C* cycles, there are *WxC* possible transient (soft) bit flips that can be injected. Because the values of *W* and *C* are large—on the order of 6400 and 10 million, respectively—and because we have multiple benchmarks, we necessarily must sample from this enormous space of possible experiments. We consider every wire, but we randomly sample a time from the first 100K cycles. We flip the value on that wire on that cycle and hold it for one cycle before letting the wire's value change.

For permanent errors, we can exhaustively evaluate coverage. For each wire, we perform two experiments: one in which we inject a stuck-at-0 and one in which we inject a stuck-at-1, both at time zero. Future work will explore other permanent fault models, including bridging faults, open circuits, timing faults, etc.

Our benchmarks are from the Spec2000 benchmark suite. We use all four of the benchmarks that the FabScalar group has made available to run on their cores and that do not perform division instructions: bzip, gzip, mcf, and parser.

B. Permanent Error Results

Our permanent error results are shown in Figure 3. The figure divides up the injected errors into four categories, based on whether the errors are masked and/or detected. The most important category is unmasked+undetected, because these are the silent data corruptions (SDCs) that we seek to avoid. If an error is unmasked (i.e., has an impact on the outcome of the software), we want Nostradamus to detect it. SDCs comprise approximately 5% of all injected errors. If we factor out the masked errors, then we see that Nostradamus detects 88% of all unmasked errors.

These results confirm that Nostradamus successfully detects a large majority of unmasked errors—but it fundamentally cannot detect all of them. The majority of the wires that were susceptible to SDCs are in Fetch, which comprises 16% of the core's wires and much of which is unprotected by Nostradamus. (In Fetch, Nostradamus protects the PC update logic and branch prediction.)

The fraction of errors that is masked is perhaps surprisingly large, but is consistent with recent work [11]. Many of these errors are in functional units, because functional units have a large number of wires, many of which are only unmasked for specific and rare combinations of inputs. Another source of unmasked errors derives from how the FabScalar core was written for clarity and ease of debugging, rather than minimizing circuitry. Thus there exist wires that are not functionally relevant and would likely be optimized away during synthesis. We considered re-running experiments on the post-synthesis circuitry, but the time required to simulate at that level of detail is prohibitive.

C. Transient Error Results

In Figure 4, we show the results for the complete set of transient error injections on one benchmark, bzip. (Results on other benchmarks were similar.) The graph classifies wires (on the x-axis) based on what fraction of injected errors leads to silent data corruptions (SDCs, on the y-axis), and the wires are sorted from lowest to highest value of SDC fraction. Across the benchmarks, approximately 6000 of the 6393 (94%) wires experience zero SDCs, with a range of 92.4% (parser) to 95.4% (gzip). For these wires, every injected error is either masked or detected by Nostradamus. The curves then rise sharply from zero to 0.5 and towards 1.0, indicating that, of the wires that are susceptible to SDCs, a sizable fraction are very susceptible to SDCs. These results show that Nostradamus successfully prevents errors from causing SDCs.

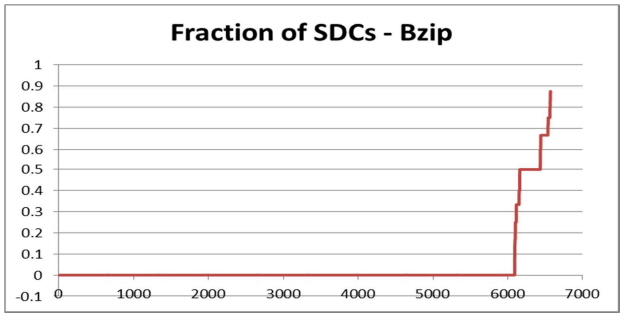

Figure 4. Transient errors on all 6,393 wires

V. Conclusions

We have developed Nostradamus, a novel error detection scheme for superscalar processor cores. We have demonstrated that Nostradamus is effective at detecting errors and that its costs are modest.

Acknowledgments

We thank Steve Raasch for helping to inspire this project and for his feedback on this work. This material is based on work supported by the National Science Foundation under grant CCF-111-5367.

References

[1] T. M. Austin, "DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design," in *Proceedings of the 32nd Annual IEEE/ACM International Symposium on Microarchitecture*, 1999.

[2] J. Carretero et al., "End-to-End Register Data-flow Continuous Self-test," in *Proc. of the 36th Annual International Symposium on Computer Architecture*, 2009.

[3] J. Carretero et al., "On-line Failure Detection in Memory Order Buffers," in *IEEE Int'l Test Conference,* 2008.

[4] N. K. Choudhary et al., "FabScalar: Composing Synthesizable RTL Designs of Arbitrary Cores Within a Canonical Superscalar Template," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, 2011.

[5] C. Constantinescu, "Using Physical and Simulated Fault Injection to Evaluate Error Detection Mechanisms," in *Proceedings of the Pacific Rim International Symposium on Dependable Computing*, 1999.

[6] M.-L. Li, P. Ramachandran, S. K. Sahoo, S. Adve, V. Adve, and Y. Zhou, "Understanding the Propagation of Hard Errors to Software and Implications for Resilient System Design," in *Proceedings of the Thirteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2008.

[7] A. Meixner, M. E. Bauer, and D. J. Sorin, "Argus: Low-Cost, Comprehensive Error Detection in Simple Cores," in *Proceedings of the 40th Annual International Symposium on Microarchitecture*, 2007.

[8] A. Meixner and D. J. Sorin, "Error Detection Using Dynamic Dataflow Verification," in *Proc. of the International Conference on Parallel Architectures and Compilation Techniques*, 2007.

[9] Nangate Development Team, "Nangate 45nm Open Cell Library." 2012.

[10] S. Nomura, M. D. Sinclair, C.-H. Ho, V. Govindaraju, M. de Kruijf, and K. Sankaralingam, "Sampling + DMR: Practical and Low-overhead Permanent Fault Detection," in *Proceedings of the 38th International Symposium on Computer Architecture*, 2011.

[11] A. Pellegrini et al., "CrashTest'ing SWAT: Accurate, Gate-Level Evaluation of Symptom-Based Resiliency Solutions," in *Design, Automation & Test in Europe,* 2012.

[12] A. Pellegrini et al., "CrashTest: A Fast High-Fidelity FPGA-based Resiliency Analysis Framework," in *Proceedings of the IEEE International Conference on Computer Design*, 2008.

[13] S. K. Reinhardt and S. S. Mukherjee, "Transient Fault Detection via Simultaneous Multithreading," in *Proceedings of the 27th Annual International Symposium on Computer Architecture*, 2000, pp. 25–36.

[14] G. A. Reis et al, "SWIFT: Software Implemented Fault Tolerance," in *Proc. of the Int'l Symp. on Code Generation and Optimization*, 2005.

[15] E. Rotenberg, "AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors," in *Proc. of the 29th International Symposium on Fault-Tolerant Computing Systems*, 1999.

[16] F. F. Sellers, M.-Y. Hsiao, and L. W. Bearnson, *Error Detecting Logic for Digital Computers*. McGraw Hill Book Company, 1968.

[17] S. Shyam, K. Constantinides, S. Phadke, V. Bertacco, and T. Austin, "Ultra Low-Cost Defect Protection for Microprocessor Pipelines," in *Proceedings of the Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.