Design and Evaluation of a Timestamp-Based Concurrent Error Detection Method (CED) in a Modern Microprocessor Controller

Michail Maniatakos[1], Naghmeh Karimi[2], Yiorgos Makris[1],
Abhijit Jas[3], Chandra Tirumurti[4]

[1] EE Department - Yale University
[2] ECE Department - University of Tehran
[3] Validation & Test Solutions - Intel Corporation
[4] Strategic CAD Labs - Intel Corporation
{michail.maniatakos, naghmeh.karimi, yiorgos.makris}@yale.edu
{abhijit.jas, chandra.tirumurti}@intel.com

Abstract

This paper presents a concurrent error detection technique for the control logic of a modern microprocessor. Our method is based on execution time prediction for each instruction executing in the processor. To evaluate the proposed method, we use a superscalar, dynamically-scheduled, out-of-order, Alpha-like microprocessor, on which we execute SPEC2000 integer benchmarks and we consider the coverage and the detection latency for faults in the scheduler module of the microprocessor controller. Experimental results show that through this method, a large percentage of control logic faults can be detected with low latency during normal operation of the processor.

1. Introduction

The rapidly shrinking feature sizes of semiconductor fabrication, along with the corresponding physical challenges that they incur, continue to give rise to various design robustness concerns. For example, the frequent occurrence of transient errors has, once again, surfaced as a problem of contemporary interest. While soft errors, occurring due to strikes by neutrons or alpha particles which potentially lead to corresponding single event upsets (SEUs) in memory bits, or single event transients (SETs) in combinational logic have received the lion’s share of attention, they only constitute part of the problem. Indeed, various other issues related to design marginalities, process variations and corner operating conditions are starting to cause errors and to play an increasingly important role. Ranging in duration from single events to permanent faults, such errors have revived interest in concurrent error detection (CED) and/or correction methods that may ameliorate or resolve their effect.

CED [1, 2] has been extensively studied in the past and numerous ideas and solutions have been investigated along various directions. The simplest approach is duplication, wherein a replica of the circuit is added to the design, possibly diversely implemented to avoid common mode failures [3]. The original and the replica serve as predictors of the functionality of each other and a simple comparator indicates any discrepancy in their outputs, thus detecting potential malfunctions. While simple, this technique is prohibitively expensive. Partial duplication solutions focusing only on the most critical parts of a circuit have, therefore, also been explored [4]. Another very popular CED approach has been the use of various codes, especially within the context of finite state

1550-5774/08 $25.00 © 2008 IEEE
DOI 10.1109/DFT.2008.59

machine (FSM) controllers. Several redesign and resynthesis methods are described in [5, 6], wherein parity or various unordered codes are employed to encode the states of the circuit. Utilization of multiple parity bits is also examined in [7] within the context of FSMs. These methods guarantee latency-free error detection; on the down side, they are intrusive and expensive. Non-intrusive CED methods have also been proposed. Implementations based on Bose-Lin and Berger codes are presented in [8] and [9], respectively, while parity-based CED methods are described in [1, 10, 11]. While the aforementioned methods guarantee latency-free detection of all errors, their cost is often prohibitive. Trading-off the incurred cost by allowing a non-zero, yet bounded latency has also been investigated [12].

At a coarser level, an attempt to identify inherent invariance either at the gate-level [13] or at the RTL [14] of a design has been made. Such invariance can be monitored during the normal operation of a circuit to identify errors that cause it to be violated. In [13] such invariance is mined from the gate-level of a controller implementation in the form of assertions, which are evaluated through simulation in order to select a cost-effective appropriate subset. The same principle governs the approach in [14]; therein, however, invariance is identified through a path-construction algorithm, which exploits inherent transparency channels that exist in the RTL description of a modular design.

At an even higher architectural level, several concurrent error detection and/or correction methods have been proposed. The concept of watchdog processors, which compute control-flow signatures and compare them to expected correct values, known at compilation time, is proposed in [15]. Concepts akin to instruction-level duplication and comparison to identify erroneous results are examined in [16, 17]. In [18], the authors examine the vulnerability of different parts of a microprocessor to soft errors and recommend various strategies (including register file protection with codes, parity coding to protect instruction words, and a timeout counter to flush the pipeline when no activity occurs for prolonged periods) to detect/correct such errors. Similar analysis is performed in [19], based on the concept of Architectural Vulnerability Factor (AVF), which prioritizes microprocessor modules based on their susceptibility. Such metrics can prove very useful in guiding allocation of CED resources

This pluralism of options implies that no one solution fits the needs of every circuit or even every part of a circuit. Furthermore, it stresses the fact that generic solutions typically incur prohibitive cost, often times without providing commensurate coverage. Therefore, while developing CED methods for a circuit, it is important to tailor the solutions by leveraging the specifics of each module.

In this work, we combine several of the key ideas proposed in the above references and we develop a CED method for the scheduler module of a modern microprocessor controller. The proposed method utilizes architectural information (i.e. the functionality of the scheduler) to construct an invariant (i.e. the relation between the dispatching and starting execution time-stamps of an instruction). Consequently, monitoring of this simple invariant during normal operation enables detection of any fault or error resulting in a discrepancy between the expected and actual timestamps, with small detection latency, often even before the error corrupts the architectural state. We note that, while the microprocessor datapath is equally important, we mainly focus on control logic for two reasons. First, CED for datapath is understood much better and various residue code-based techniques have been successfully applied. Second, advanced architectural features complicate significantly the task of the controller, making it much harder to analyze or predict its behavior in the presence of errors.

The remaining of this paper is organized as follows. In Section 2, we briefly review a fault simulation infrastructure that we have previously developed around a modern microprocessor and which we use to evaluate the proposed CED method. In Section 3, we discuss the details of the timestamp-based CED method as well as the Scheduler module of the microprocessor controller, which is targeted by this method. In Section 4, we experimentally assess the coverage and the detection latency of the proposed CED method using the aforementioned infrastructure to perform extensive experiments while the target microprocessor executes typical SPEC benchmark programs. Conclusions and future directions are provided in Section 5.

2. Background

The research work described herein builds upon a previously developed infrastructure, which is presented in detail in [20]. The employed model is the Illinois Verilog Model (IVM) [18], an Alpha 21264-like microprocessor featuring superscalar, out-of-order execution. The complexity of such a model reflects most of the features of modern, high-performance microprocessors enabling accurate evaluation of CED techniques targeting the same. The developed infrastructure supports simulation of actual programs (i.e. SPEC benchmarks), injection and simulation of Register Transfer Level (RTL) faults (both stuck-at and transients), as well as extensive I/O capabilities (i.e. Machine State Dumping and Trace Dumping features).

Using the developed infrastructure, in [20] we investigated the correlation between RTL faults in the control logic and their instruction-level impact on the execution flow of typical programs. Specifically, we injected stuck-at faults at the Scheduler and the Reorder Buffer (ROB) modules of the microprocessor and we studied their impact on the execution of integer SPEC benchmarks. The arising Instruction Level Errors (ILEs) were divided into five groups reflecting the key aspects of instruction execution in a superscalar out-of-order microprocessor, namely (i) the operation that is executed, (ii) the operands that are being used, (iii) the functional unit where execution takes place, (iv) the starting and finishing time of execution, and (v) the order of commitment. These groups were further divided in 13 Types, as detailed in Table 1.

Table 1. Instruction level error types

Group 1: Operation Errors	**Type 1:**	Incorrect (yet valid) operation code used
	Type 2:	Invalid operation code used
Group 2: Operand Errors	**Type 3:**	Incorrect (yet valid) register addressed
	Type 4:	Invalid register addressed
	Type 5:	Premature use of register contents
	Type 6:	Incorrect immediate operand used
Group 3: Execution Errors	**Type 7:**	Incorrect functional unit type utilized
	Type 8:	Multiple functional units utilized
Group 4: Timing Errors	**Type 9:**	Early commencement
	Type 10:	Late or no commencement
	Type 11:	Longer duration
	Type 12:	Shorter duration
Group 5: Order Errors	**Type 13:**	Commitment order violation

Using RTL fault injection, 16,904 stuck-at faults were injected at the Scheduler module for each of the SPEC benchmarks executed. For every injected fault, an automated cycle-by-cycle analysis of the Scheduler traces was used to classify the fault to one of the ILE types defined in Table 1. The classification results are presented in Figure 1, averaged among the executed SPEC benchmarks.

The results of the study show that Timing Errors were the most dominant group of errors, particularly Type 10 (Late or no instruction commencement) and Type 11 (Longer instruction duration). Based on this observation, in this work we develop a timestamp-based CED technique which specifically targets these erroneous behaviors.

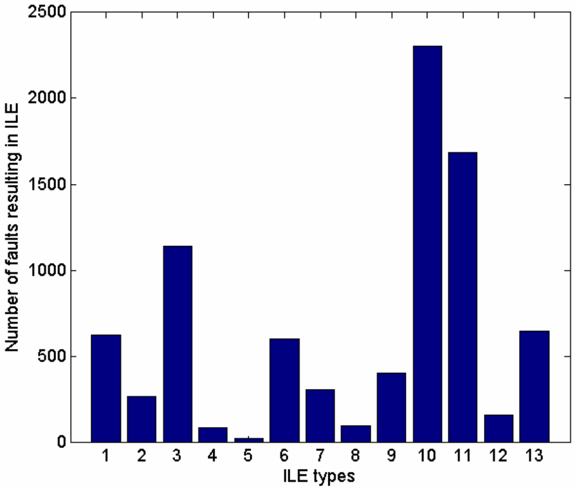

Figure 1. Classification of stuck-at faults of Scheduler module into ILE Types

3. Developed CED method

In this section, we propose a cost-effective strategy for Concurrent Error Detection (CED) of the control logic of the above processor. Using this strategy, transient faults, as well as permanent faults occurring due to gradual degradation during the lifetime of the microprocessor can be detected during its normal operation. The following subsections present the structure of the targeted Scheduler module and the CED technique.

3.1. Scheduler module

The targeted Scheduler is a dynamic module which can issue up to 6 instructions in each clock cycle. Instructions are issued out of order depending on the following factors:

- Availability of the instructions in the Scheduler module
- Avoidance of data hazards
- Avoidance of structural hazards

The Scheduler module contains an array of up to 32 instructions waiting to be issued. Each instruction coming to the Scheduler, resides in this buffer until an acknowledgement is received from the execution unit that it can start execution. At this time, the corresponding location in the scheduler list can be used for another newly arriving instruction to the scheduler module.

Structural hazards are considered by the Scheduler before issuing an instruction. The microprocessor model has 2 simple, 1 complex, 1 branch and 2 memory instruction functional units. Thus there is a limitation on the number of instructions of each type that can be issued in each clock cycle.

The microprocessor model also includes a Rename module. The renaming process removes the possibility of Write-After-Read (WAR) and Write-After-Write (WAW) hazards. However, due to the dependency of the instruction operands there can still be a Read-After-Write (RAW) hazard. To deal with such RAW hazards, the Scheduler module uses a Scoreboard technique [21].

Furthermore, for each instruction coming to the Scheduler, the Reorder Buffer module assigns an identification number, called *ROBid*. This *ROBid* follows the instruction until it commits and serves as a mechanism for ensuring in-order instruction commitment in the out-of-order execution of the microprocessor.

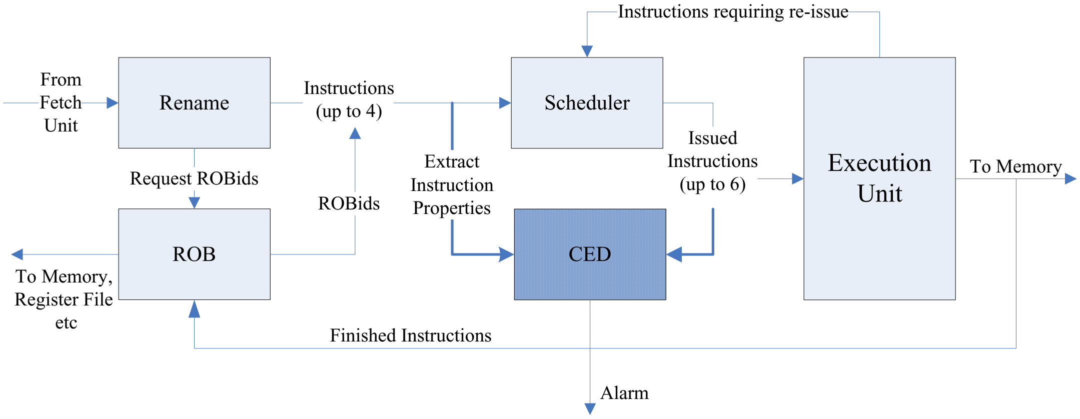

Figure 2. Block diagram of proposed CED technique

3.2. CED strategy

Based on the functionality of the Scheduler module discussed above, we implemented a CED mechanism to detect a portion of permanent as well as transient errors of this module, during the normal operation of the microprocessor. Our method is based on execution time prediction for each instruction resides in Scheduler module. The prediction is based on the fact that each coming instruction to the Scheduler starts execution after a certain number of clock cycles assuming that there is no structural or data hazard. A high-level block diagram of the proposed method is shown in Figure 2.

The CED mechanism keeps track of the incoming instructions to the Scheduler and predicts the execution time of these instructions based on the information gathered from the Scoreboard and Scheduler units. If an instruction encounters no data or structural hazards due, its execution time is predicted considering the structure of the microprocessor model. Then the *ROBid* of that instruction, which uniquely identifies it, is stored along with the predicted execution time.

The CED module checks if instructions stored in its internal buffer are correctly executed at the appropriate functional unit at the correct timestamp. An instruction may be replayed if the operands are not ready; this happens in the analyzed microprocessor model because forwarding mechanisms may provide the necessary operands directly to the execution unit, so the scheduler tries to issue instructions even if their operands are not ready yet. The developed CED technique checks the Scoreboard module, which is part of the Scheduler, for the availability of the operands and predicts the starting timestamps.

The CED algorithm can be summarized as follows:

1. Extract *ROBid*, Instruction Type and Operands information from instruction array entering the Scheduler.
2. Based on bookkeeping information of functional units utilized and operands in use, predict when an instruction should start execution.

3. Track instruction execution at the functional units. Raise alarm if any discrepancy identified.

The proposed CED method is not duplication, since it reuses parts of the Scheduler such as the Scoreboard, which can be protected easily by using techniques such as parity.

4. Experimental Results

In this section, we discuss the fault model used to evaluate the developed CED method and we present an extensive analysis of the results. Specifically, we report statistics about the fault coverage and the latency of the proposed CED method.

4.1. Experimental setup

To asses the fault coverage of the developed CED, the stuck-at fault model is used. Because our target is control logic, and more specifically the scheduler, faults are injected only in this module. A total of 16,904 s@0 and s@1 faults are injected using the RTL model fault injection technique of [20], which was briefly described in section 2.

In order to evaluate the developed CED method, six different SPEC benchmarks are utilized. Each benchmark is executed at the fault-injected RTL model for 20,000 cycles. After the end of the simulation, the architectural state of the microprocessor is compared to a fault-free (golden) run. If any discrepancies are identified, then the error is classified into one of two different sets: i) if the error propagates to the architecture register file, then the execution is marked as Erroneous, and ii) otherwise, if a discrepancy exists in the machine state but not in the architecture register file, the fault is likely masked and propagates to a part of the microprocessor that does not affect the execution – thus we classify the execution as Masked.

Besides architectural state discrepancies, an injected fault may lead to a different simulation outcome: stall of the pipeline, if the executed program uses unimplemented instructions. As explained in [20], the microprocessor model lacks certain instructions, such as system calls or floating-point operations. Even though the golden run is carefully chosen so that no such instructions are fetched, a fault may still drive the microprocessor to incorrectly call one of these instructions in the same time window. In this case, due to the described microprocessor model limitations, the execution stalls and the corresponding run is marked as Stalled. This is indeed an erroneous behavior, yet we cannot tell whether the proposed CED method will detect the fault after the unimplemented instruction. Thus, the number reported is very pessimistic and covers faults detected before the microprocessor stalls; in a full instruction set implementation the fault coverage would be higher. The complete classification process is presented in Figure. 3.

Besides the fault coverage metric, detection latency is also reported and analyzed. Latency is defined as the difference between the time that the CED error signal is activated and the time when a discrepancy first appears in the architecture register file of the microprocessor. Sometimes, the CED output signal may be activated before a discrepancy appears. We call this case an early detection. This type of detection provides an easy recovery, since the register file is unaffected. Similarly, if the CED fires after a discrepancy is identified, we call it a late detection. In this case, the microprocessor should rollback to a previous valid checkpoint.

4.2. Experiment results and discussion

The most important aspect of a CED method is the attained fault coverage. Our fault coverage analysis consists of two different parts: i) fault coverage of the cases where the architectural state is different at the end of the simulation, and ii) fault coverage of the cases where a pipeline stall occurs. As explained in section 4.1, the latter is a pessimistic estimate because there is no capability to evaluate the CED behavior after the stalling point.

Figure 4 presents the classification of the 16,904 runs for each SPEC benchmark. The Erroneous and Stalled classes are the target of our CED method. The differences between the numbers of corrupted runs for each benchmark exist because each benchmark uses a different set of instructions during the simulation window, which may or may not use the stuck-at bit. Nevertheless, this variability provides a better estimate of the CED performance, since different instructions are utilized for each SPEC benchmark.

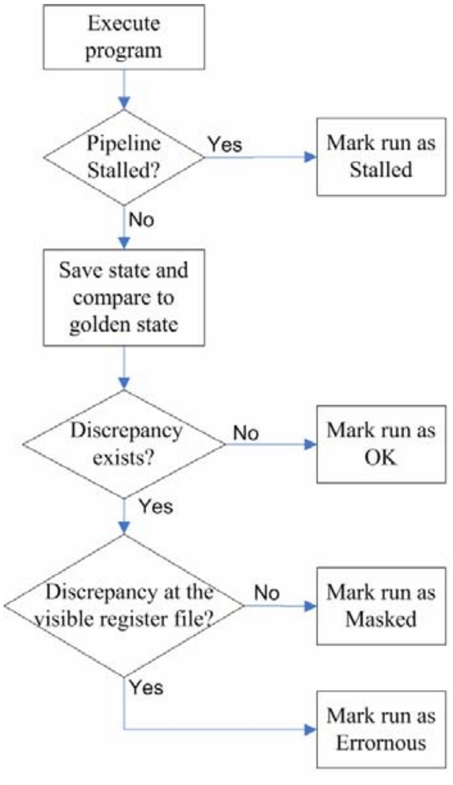

Figure 3. Simulation outcome classes

Proceeding to the actual results, Figure 5 presents the fault coverage percentage for each of the aforementioned classes (Erroneous and Stalled). The CED detects 52.2% of the Erroneous cases on average. The consistency of the fault coverage among the 6 benchmarks corroborates that the CED method is independent of the program load.

Figure 6 shows the fault coverage of applying our CED method for each ILE type discussed in section 2. Even though the proposed CED method targets timing issues (Types 9 and 10), fault coverage in these groups is close to 50%. This happens because, as described in [20], errors that do not fall into one of the 13 Types usually appear as timing errors. These errors cannot be targeted by our CED method, as individual instruction timing appears to be correct.

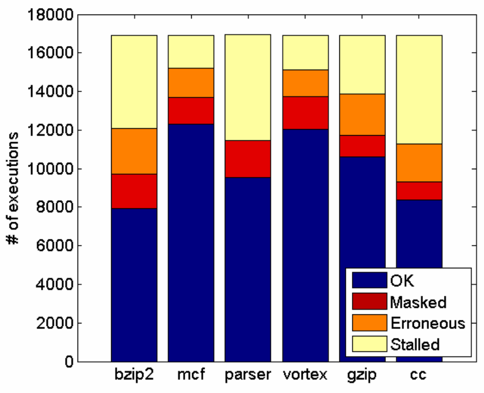

Figure 4. Simulation outcome results

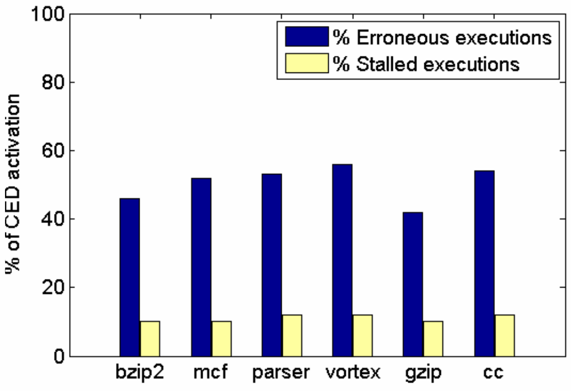

Figure 5. Fault coverage results

Another interesting conclusion drawn from Figure 6, is that the CED method excels at detecting faults concerning utilization of Functional Units (FUs) and In-Order Instruction Commitment. This is expected due to the nature of the CED method and its location between the Scheduler, the ROB and the Execution Unit. The CED checks the validity of the FUs utilization by checking the opcodes and the operands. In addition many commitment order violation (Type 13) errors are the result of timing errors.

As defined in section 4.1, detection latency is the difference between the time that the CED error signal is activated and the time when a discrepancy first appears in the architecture register file of the microprocessor. Early detection implies that the CED alarm signal is triggered before the discrepancy appears; otherwise we have a late detection.

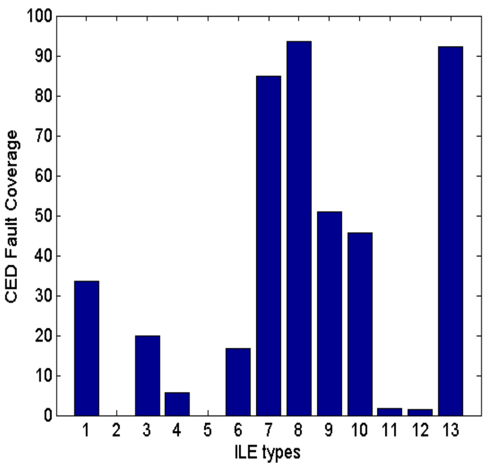

Figure 6. Correlation of CED detections to ILEs

The results presented in Table 2 show that, in most cases, an early detection is reported. Latency may not be reported for all detections, because a discrepancy may not appear at the programmer-visible register file within the time window of 20,000 cycles. As can be observed, an average of 95% of the total number of reported latencies are early detections (latency <= 0).

The normalized average latency presented in the last column of Table 2 is the average of latencies, with an early detection considered as latency of 0 cycles. The early detection property of the CED indicates that, in most cases, a pipeline flush and restart is enough to correct the fault. For the late detection cases, the worst case latency does not exceed 35 cycles, according to Table 2, so a checkpoint and restore operation could be fine-tuned accordingly to correct the microprocessor state.

Table 2. Fault detection latency results

SPEC Benchmark	Early detection	Normalized Average Latency (in clock cycles)
bzip2	93%	17
mcf	98%	22
parser	96%	5
vortex	97%	22
gzip	95%	34
cc	92%	35
Average	**95%**	**23**

5. Conclusion and Future Work

The CED method proposed in this work for control logic of modern microprocessors is based on the observation that the impact of most low-level faults in a modern microprocessor is quickly visible on the starting and finishing instruction execution timestamps. Predicting these timestamps and checking against their actual execution values during normal operation results in detection of over 52% of the faults in the Scheduler module with an average detection latency of 35 cycles. We are currently extending the scope of the CED method to other control logic units. Our ultimate goal is to efficiently detect faults in the control logic of a microprocessor by adding an ensemble of small CED techniques, each targeting a different set of faults.

Acknowledgement

This research was sponsored by a generous gift by Intel Corporation and was performed while the second author was a visiting student at Yale University.

6. References

[1] M. Goessel, and S. Graf, Error Detection Circuits, McGraw-Hill, 1993.

[2] S. Mitra, and E. J. McCluskey, “Which Concurrent Error Detection Scheme to Choose?” in Proc. of the International Test Conference, 2000, pp. 985–994.

[3] A. Avizienis, and J. P. J. Kelly, “Fault Tolerance by Design Diversity: Concepts and Experiments”, IEEE Transactions on Computers, vol. 17, no. 8, pp. 67-80, 1984.

[4] K. Mohanram, and N. A. Touba, “Cost-Effective Approach for Reducing Soft Error Rate in Logic Circuits,” in Proc. of the International Test Conference, 2003, pp. 893–901.

[5] G. Aksenova, and E. Sogomonyan, “Design of Self-Checking Built-in Check Circuits for Automata with Memory,” Automation and Remote Control, vol. 36, no. 7, pp. 1169–1177, 1975.

[6] S. Dhawan, and R. C. D. Vries, “Design of Self-Checking Sequential Machines,” IEEE Transactions on Computers, vol. 37, no. 10, 1988, pp. 1280–1284.

[7] C. Zeng, N. Saxena, and E.J. McCluskey, “Finite State Machine Synthesis with Concurrent Error Detection,” in Proc. of the International Test Conference, 1998, pp. 672–679.

[8] D. Das, and N. A. Touba, “Synthesis of Circuits with Low-Cost Concurrent Error Detection Based on Bose-Lin Codes,” Journal of Electronic Testing: Theory and Applications, vol. 15, no. 2, 1999, pp. 145–155.

[9] N. K. Jha and S.-J. Wang, “Design and Synthesis of Self-Checking VLSI Circuits,” IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 12, no. 6, 1993, pp. 878–887.

[10] R. A. Parekhji, G. Venkatesh, and S. D. Sherlekar, “Concurrent Error Detection Using Monitoring Machines,” IEEE Design and Test of Computers, vol. 12, no. 3, 1995, pp. 24–32.

[11] S. Almukhaizim, P. Drineas, and Y. Makris, “Entropy-Driven Parity-Tree Selection for Low-Overhead Concurrent Error Detection in Finite State Machines,” IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 25, no. 8, 2006, pp. 1547-1554.

[12] S. Almukhaizim, P. Drineas, and Y. Makris, “On Concurrent Error Detection with Bounded Latency in FSMs in Proc. of the IEEE Design Automation and Test in Europe Conference (DATE), 2004, pp. 596-601.

[13] R. Vemu, A. Jas, J. A. Abraham, S. Patil, and R. Galivanche, “Low-Cost Concurrent Error Detection Technique for Processor Control Logic,” in Proc. of Design Automation and Test in Europe, 2008, pp. 897-902.

[14] Y. Makris, I. Bayraktaroglu, and A. Orailoglu, “Enhancing Reliability of RTL Controller-Datapath Circuits via Invariant-Based Concurrent Test,” IEEE Transactions on Reliability, vol. 53, no. 2, 2004, pp. 269-278.

[15] A. Mahmood, and E. J. McCluskey, “Concurrent Error Detection Using Watchdog Processors-A Survey”, IEEE Transactions on Computers, vol. 37, no. 2, 1988, pp. 160-174.

[16] A. Mendelson, and N. Suri, “Designing High-Performance and Reliable Superscalar Architectures-the Out of Order Reliable Superscalar (O3RS) Approach”, In Proc. of International Conference on Dependable Systems and Networks, 2000, pp. 25-28.

[17] A. K. Somani, and J. Nickel, “REESE: a Method of Soft Error Detection in Microprocessors”, in Proc. of International Conference on Dependable Systems and Networks, 2001, pp.401-410.

[18] N. J. Wang, J. Quek, T. M. Rafacz, S. J. Patel, “Characterizing the Effect of Transient Faults on a High-Performance Processor Pipeline”, in Proc. of International Conference on Dependable Systems and Networks, Florence, Italy, 2004, pp.61-70.

[19] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin, “A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor,” In Proc. of International. Symposium on Microarchitecture, 2003, pp. 29-40.

[20] N. Karimi, M. Maniatakos, A. Jas, and Y. Makris, “On the Correlation between Controller Faults and Instruction-Level Errors in Modern Microprocessors,” in Proc. of International Test Conference, 2008.

[21] J. L. Hennessy, and D. A. Patterson, Computer Architecture: A Quantitative Approach, Third Edition, Morgan Kaufmann Publishers, 2003.