A low-cost concurrent error detection technique for processor control logic

Ramtilak Vemu† Abhijit Jas‡ Jacob A. Abraham† Srinivas Patil‡
Rajesh Galivanche‡

† Computer Engineering Research Center
University of Texas at Austin
{rvemu,jaa}@cerc.utexas.edu

‡ Design and Technology Solutions
Intel Corporation
{ajas,spatil,rgalivanche}@intel.com

Abstract

This paper presents a concurrent error detection technique targeted towards control logic in a processor with emphasis on low area overhead. Rather than detect all modeled transient faults, the technique selects faults which have a high probability of causing damage to the architectural state of the processor and protects the circuit against these faults. Fault detection is achieved through a series of assertions. Each assertion is an implication from inputs to the outputs of a combinational circuit. Fault simulation experiments performed on control logic modules of an industrial processor suggest that high reduction in damage causing faults can be achieved with a low overhead.

1 Introduction

Transient faults can occur in a processor as a result of electrical noise, like crosstalk, or high energy particles, like neutrons and alpha particles. These faults can cause a program running on the processor to behave erratically, if they propagate and change the architectural state of the processor. These faults can occur in memory arrays, sequential elements or in the combinational logic in the processor. Protection against transient faults in combinational logic has not received much attention traditionally because combinational logic has a natural barrier stopping the propagation of the faults [4]. Three masking factors - logical, electrical and latching-window, reduce the probability that a transient fault propagates and latches on to sequential elements. With the current trends in the processor industry, however, the masking provided by these factors is reducing [5]. Reduced logical depth between latches means that there are more sensitized paths and hence more paths for a transient fault at a gate to propagate and latch on. Decreasing feature sizes and lowering operating voltages result in the lesser charge stored at any node. Thus the electrical noise or the energy of the particle strikes required for triggering a transient fault is decreasing. High operating frequencies mean that there are more latching-windows per unit time, thus increasing the probability that a transient fault gets latched on. Due to the reasons mentioned, the combinational portion of the processor is projected to become a dominant source of failures due to transient faults [9].

Various techniques have been proposed to detect transient faults in combinational circuits. Residue codes have been found to be very effective in detecting faults in datapath circuits. For control logic circuits, codeword based schemes have been proposed. Codes like parity [8], [11], [12], Berger [3] and Bose-Lin [1] are predicted for the outputs of the circuit and the codes of real-time outputs are matched against the predicted codes. These techniques for control logic circuits are viable for mission critical applications where reliability is of primary concern and area, timing and power take second place. There is another class of techniques which do not attempt to detect all the modeled faults like the above methods. Rather, they try to detect most of the errors at a reasonable overhead. Such techniques are more viable for mainstream applications which do not have the same stringent FIT (Failure In Time) requirements as the mission critical applications [5]. The work presented in this paper falls under this category of techniques.

This paper presents a new low-cost technique for concurrent error detection (CED) in processor control logic. The proposed technique takes advantage of the fact that transient faults in gates along some paths are much more likely to propagate to an architectural state under normal running of the processor than others and protects against errors in these paths. The technique automatically extracts the control conditions (input value combinations) under which these paths are sensitized and converts these conditions into assertions. Each assertion is an implication from the control conditions to the value of an output. Depending on the area overhead budget and the required transient error reduction, a subset of the extracted assertions can be selected for CED. The work presented here is similar to the work presented in [2] in the sense that outputs are predicted for a few input combinations. As opposed to this technique, the present work

978-3-9810801-3-1/DATE08 © 2008 EDAA

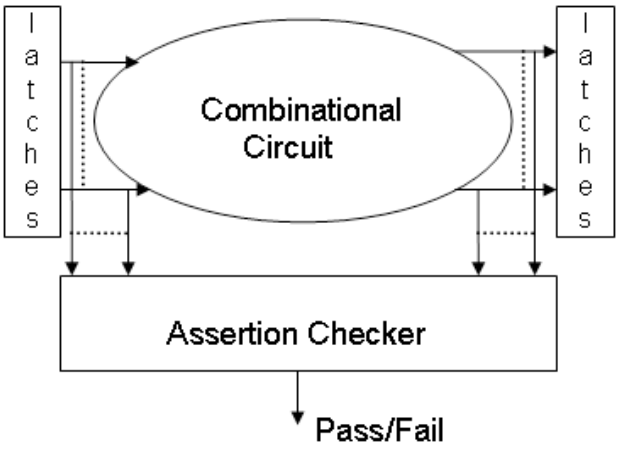

Figure 1: Block diagram for the proposed CED

does not make any assumptions as to the duration of transient faults and has a very low latency. The work presented here can be considered a more fine grained approach than the concept of architectural vulnerability factor (AVF) [10]. Instead of just determining which modules are more vulnerable, we determine which flops in these modules are the most vulnerable and protect the combinational logic which feeds these flops.

Fault simulation experiments were performed on selected control logic modules in an industrial processor using sample segments of real applications. The proposed technique was implemented for each of these modules for fault escape reductions of 50%, 75%, 90%, 95% and 99%. A fault escape happens if a transient fault propagates undetected to the architectural state of the processor. Results show that high fault escape reductions can be achieved at low costs. Over 95% fault escape reduction can be obtained with just 25% area overhead.

The rest of the paper is organized as follows. Section 2 gives an overview of the proposed technique and provides an insight as to why high fault escape reductions can be obtained with a few assertions. Section 3 and Section 4 provide a detailed discussion about the algorithm. Section 5 presents the experimental results and Section 6 provides conclusions.

2 Overview

In this section, we will present an overview of the proposed technique. The technique protects the combinational portion of the circuit against transient errors. Figure 1 shows a circuit which has CED capability. The technique introduces an *Assertion Checker* which takes as inputs the inputs and the outputs of the combinational circuit and which gives out a signal whether they conform to a series of assertions. Each assertion is an implication of the form $antecedent \implies consequent$ (if antecedent is true, then consequent is true). The antecedent in each assertion is a minterm on a subset of inputs to the circuit. All the inputs need not be part of the antecedent. In many cases, the antecedent is a minterm on just one or two of the inputs. The consequent in each assertion is a literal of an output of the circuit. For example, the following would be a valid assertion according to our technique: $i2i3' \implies o5$, where $i2$ and $i3$ are two of the inputs to the circuit and $o5$ is an output of the circuit. This assertion states that $o5$ should have a value of 1 when $i2 = 1$ and $i3 = 0$. An assertion will detect all the faults which propagate to the output in the consequent when the antecedent is true. The above example assertion will detect all faults propagating to $o5$ when $i2, i3 = 1, 0$.

Table 1: Distribution of input vectors of combinational portion of an example module

Number of unique vectors	% of total vectors (703547 vectors)
32	50
397	75
5392	90
37175	95
65317	99
72353	100

In testing terminology, the antecedent of an assertion would form a test vector for a stuck-at fault for the output in the consequent. In the above example, $i2 = 1$ and $i3 = 0$ would form a test vector for a stuck-at-0 fault at $o5$. In fact, any test vector which detects any stuck-at fault at an output of the combinational circuit can be converted into an assertion on that output.

An assertion can also be viewed as checking for a subset of the truth table for the corresponding output. The above example checks for that subset of the $o5$ truth table which has $i2 = 1$ and $i3 = 0$.

In order to keep the overhead for concurrent error detection to a minimum, we need to select the minimal set of assertions such that the required transient fault coverage is achieved. To assist us in selecting the assertions to be included in the assertion checker, we use transient fault simulations using sample segments of real applications and take into consideration only the faults which propagate all the way to the architectural states of the processor. This differs from the experimental methodology followed in almost all previous CED schemes proposed [1], [3], [5], [8], [11], [12], where random vectors are used as inputs of the combinational circuits and all the faults which propagate to the outputs of the combinational portion are considered important. The methodology followed in this paper is similar to the one used in [6] for logic derating.

The effectiveness of an assertion in detecting transient faults which propagate to the primary outputs varies widely depending on the following factors.

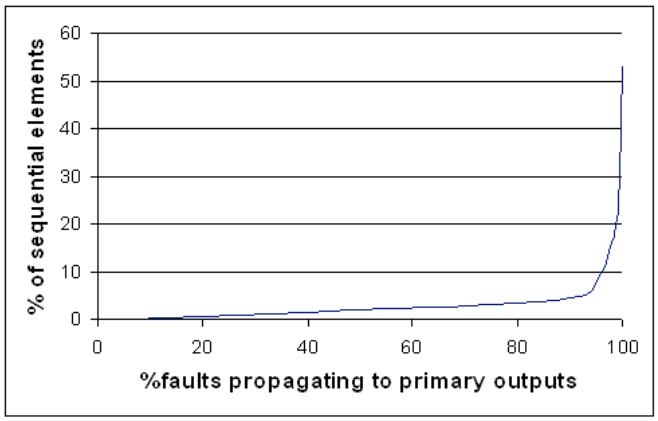

Figure 2: Asymmetry among flops contributing to faults which propagate to primary outputs

- Faults in some paths are more likely to propagate and latch on to sequential elements than others. In the control-logic of a processor, the distribution of vectors applied at run-time to the inputs of the combinational portion (primary inputs as well as outputs of latches/flops) is highly skewed. A small subset of input vectors is applied for a large percentage of clock cycles. This is due to the fact that some state transitions in the finite state machine (FSM) of a control logic module are more common than others. Additionally, some input combinations are invalid and hence cannot occur. As a result, some paths in the circuit are more often exercised than others. Transient faults in these paths are more likely to propagate to the outputs of the combinational part of the circuit and hence to the inputs of the sequential elements (latches and flops). To show their skewed nature, we collected vectors that were applied at the inputs of the combinational portions of a control logic module (*module3* in Table 3) during 703547 cycles. The module has 390 inputs to the combinational portion. We collected these vectors from traces of sample programs running on the processor. The unique vectors are sorted according to the number of times they occur and their distribution is shown in Table 1. The 703547 vectors have a total of 72353 unique vectors. The asymmetrical nature of the vectors can be seen from the table. Just 32 unique vectors contribute to about 50% of all the vectors.
- Due to clock gating, bit-flips at inputs at some of the sequential elements are more likely to get latched on than others.
- Bit flips in certain sequential elements are more likely to affect the architectural states of the processor than others. Bit flips in latches (flops) may be masked logically in the combinational portion which prevents them from propagating to the architectural states. Due to asymmetry in the vectors applied at the inputs of combinational logic, bit flips in some latches are more likely to propagate to the next level of latches. For example, if the output of a latch is fed to an AND gate whose other input is predominantly 0, a bit flip in that latch has a very low probability of propagating. The greater the pipeline distance between a latch and the architectural states, the more probable it is that a bit flip in that latch is masked. To show the asymmetry in the importance of latches in terms of bit flips in them propagating, we randomly injected transient faults in the sequential elements of the modules listed in Table 3. We then marked the faults which propagate to the primary outputs of the module in which each of those modules is instantiated. We sorted the sequential elements according the number of faults in the elements which propagate to the architectural states and Figure 2 shows the results. We can see that bit flips in just 5% of the flops contribute to more than 90% of all the faults propagating to the primary outputs.

The net effect of the above observations is that some assertions detect more transient faults propagating to the architectural states of the processor than others. An effective assertion is one whose consequent is on a combinational output which feeds a vulnerable latch. The antecedent of the assertion will cover the most common subset of the output truth table. The next few sections deal with how we automatically extract such assertions based on fault simulations on the circuit.

3 Algorithm for assertion extraction

This section describes the algorithm for extracting all the assertions which are valid for a particular input vector. In the next section, we integrate this algorithm with the rest of the flow for finding the minimal set of assertions. For ease of explanation, we define the term *control assignment* (CA). For any particular vector, the CA for any net in the circuit defines the assignments of values to inputs which guarantee the current value of the net (value of the net when the current input vector is applied). In testing terminology, each assignment of values can be considered a different controllability condition which is true for the current vector. A CA is in sum of products (SOP) format where each product defines a different controllability condition. A CA of $i1 + i2'$ for a net means that the net is guaranteed to have the current value if $i1 = 1$ or if $i2 = 0$. Additionally, $i1$ and $i2$ have values 1 and 0 in the current vector. A product in the control assignment for an output of the combinational circuit is a test vector for stuck-at fault at that output since propagation condition is also met (in addition to controlling condition).

If we know the CAs of all the inputs of any gate, the CA of its output can be calculated according to the rules stated below.

Table 2: Propagation of CAs for an AND gate

$i_1 i_2$	00	01	10	11
CA_o	$CA_1 + CA_2$	CA_1	CA_2	$CA_1.CA_2$

- If the gate has at least one controlling input, the CA of the output is the sum of CAs of all the gate inputs which have controlling values.
- If the gate has all non-controlling values at the inputs, the CA of the output is the product of CAs of all the gate inputs.

Table 2 illustrates the propagation of CAs for an AND gate with inputs $i1$ and $i2$. CA_1, CA_2 and CA_o represent the CAs of $i1$, $i2$ and the output of the gate. The propagation tables for other types of gates can be similarly obtained.

We will now present an algorithm for extracting the assertions on the outputs of the circuit for a given vector. Initially, all the nets in the circuit are ordered topologically (all inputs of a gate are listed before the output of the gate). For each net in the ordered list

1. If the net is an input of the circuit, the CA of the net is the positive literal of the net if the net has a value 1 in the simulation vector. The CA is the negative literal of the net if it has a value 0.
2. If the net is not an input to the circuit, calculate the CA of the net from the CAs of the inputs of the gate driving the net according to the rules stated above.
3. Convert the CA into the SOP format if it is not already in the format.
4. Trim the CA to contain only those products which have number of literals lesser than $(n+thresh)$, where n is the minimum number of literals in all the products and $thresh$ is a parameter of the algorithm.

Assertions on the outputs of the circuit are then extracted as follows. Each product in the output CA can be made an antecedent of a different assertion on that output. If the output has a value 0, then the consequent of the assertions will be the negative literal of the output. It will be the positive literal of the output otherwise.

We need to trim the CAs of the nets (step 4 in the algorithm above) to prevent the explosion of the number of terms in the CAs calculated subsequently from this CA. We trim away the products which have a large number of literals. The intuition behind this trimming is that the lesser the number of literals in the antecedent of any assertion, the more probable it is to occur very often and hence the more probable it is to be picked among the most effective assertions. On the other hand, trimming away some of the products may lead to dropping some of the assertions which may detect a large number of transient faults.

Figure 3 gives an example circuit and shows how the control assignments are propagated. Each net in the circuit is accompanied by the tuple (net name, value, control assignment). The circuit has inputs a, b, c and d and has an output y. A vector 0001 is applied to the circuit. The calculated control assignments are given in the figure. The AND gate gives an example of how CAs are propagated when both gate inputs are controlling. The output gate gives an example of how CAs are propagated when both gate inputs are non-controlling. The output y has a CA of $a'd + b'd$. Two assertions can then be extracted for the given vector, $(a'd \implies y')$ and $(b'd \implies y')$.

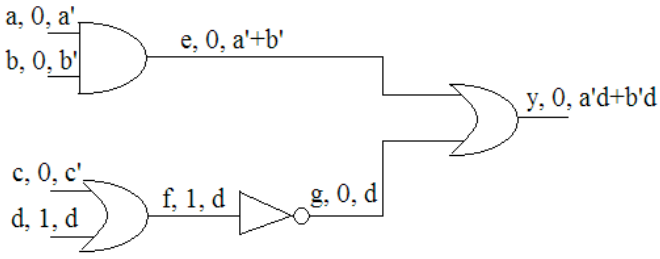

Figure 3: Example control assignment propagation

4 Algorithm for low-cost CED

In this section, we describe the algorithm for constructing the assertion checker for a given circuit. The algorithm takes as inputs the description of the circuit and the functional vectors applied to the circuit. The algorithm works for a given target reduction in fault escapes. A fault escape is a fault which propagates to the architectural state of the processor without being detected. In the absence of any concurrent error detection (CED), all the faults which propagate to the architectural states are fault escapes. In presence of CED, some of these faults are detected and hence there is a reduction in the number of fault escapes. The target reduction in fault escapes that is required is given as a parameter to the algorithm. Given below are the steps involved in implementing the algorithm.

Step 1: Performing fault simulations and building fault database. In this step we inject m transient faults in each cycle of the functional vectors, where m is a parameter to the algorithm. The transient faults are injected in the combinational portion of the design according to any given fault model (single-event transients, cross-talk faults etc.). We set as observation points the architectural state as well as the outputs of combinational portions. For each fault which propagates to the architectural state, we note the outputs of the combinational portion to which the fault propagates before being first latched on to a sequential element. For these faults, we store the vector, the fault site and the outputs of the combinational portion to which the fault propagates in a fault database. Since we store only the faults which propagate to the architectural state, we automatically consider the various masking factors mentioned in Section 2.

Step 2: Extracting assertions. For each unique vector in the fault database, we extract assertions as described in

Table 3: Details of modules used for evaluation

Module	Num. of Combinational Gates	Num. of Sequential Elements	Num. of Inputs	Num. of Outputs
module1	1509	378	64	108
module2	1314	363	79	108
module3	1692	435	100	194
module4	495	177	49	27
module5	2602	773	76	110

Section 3 for all combinational outputs to which any fault injected in that vector propagates to.

Step 3: Building the assertion database. For each extracted assertion, we find out all the faults which are detected by that assertion. An assertion detects a fault if the antecedent of the assertion holds true for the vector in which the fault is injected and the fault propagates to the output in the consequent of the assertion. We store the list of assertions and the faults they detect in an assertion database.

Step 4: Picking top assertions for a given reduction in fault escapes. Ideally we would like to pick the minimal number of assertions for detecting a given number of faults. This problem is similar to the set-covering problem and is NP-complete. We employ a simple greedy approximation algorithm to pick assertions for a given reduction in fault escapes. The target number of faults required to be detected for achieving the target reduction in fault escapes is calculated. The assertions are greedily picked till the target number of faults is detected.

Step 5: Constructing the assertion checker. Once the assertions needed for a given reduction in fault escapes are picked, the assertion checker is constructed by synthesizing the conjunction of all the individual assertions. We considered two different implementations for synthesizing the assertion checker - a totally self-checking checker and a self-exercising strongly code-disjoint checker [7]. A dual-rail implementation is used for synthesizing the self-checking checker. For the self-exercising checker, during the test phase, all the antecedents are forced to be true and the consequents are forced to be false one after the other. This implementation takes advantage of the fact that the assertion checker is the conjunction of all the individual assertions to obtain a low-overhead self-exercising checker.

5 Experimental results

The algorithm presented in this paper was evaluated on five random control logic modules in the integer execution unit of an industrial processor. The modules - $module1$, $module2$, $module3$, $module4$ and $module5$ - are instantiated in the execution unit of the processor. The details of these modules are given in Table 3.

An in-house transient fault simulator was used for all the fault simulations. The vectors used for fault simulation are functional traces extracted when running programs on the processor simulation model. 416 different functional traces with a total of 703547 vectors are used. For each transient fault to be injected a fault site was chosen randomly among all the nets in a module and the value at the net during a given cycle was corrupted. 5 transient faults were injected per cycle (3.5 million faults) for each module considered. A fault was considered to escape if it propagates to the primary outputs of the execution unit. An implicit assumption here is that the faults which propagate to the primary outputs of the execution unit are going to affect the architectural state of the processor.

The entire algorithm for extracting and picking assertions is written in perl. The program was run 5 times for each module with target fault escape reductions of 50%, 75%, 90%, 95% and 99%. Synopsys Design Analyzer was used to synthesize the modules. The assertion checkers (both self-checking and self-exercising) for each point were implemented and the area overheads were calculated. The technology library used is the lsi_10k library distributed along with Synopsys Design Compiler. For comparison purposes, the partial duplication technique described in [5] was also implemented on all five of the modules for the given target fault escape reductions. Consistent with our methodology, we considered only the faults which escape instead of considering all the faults which propagate to outputs of combinational logic.

Table 4 shows the area overhead results for partial duplication (PD), the proposed technique with dual rail implementation (PT-D) and with self-exercising implementation (PT-S) when achieving different fault escape reductions. The average area overheads for different fault escape targets are plotted in Figure 4. It can be seen that high amount of fault escape reductions can be obtained with a low area overhead. On an average, 50% fault escape reduction can be obtained with just 3% overhead. This number increases to 54% for PT-D and 42% for PT-S when the target fault escape reduction is 99%. It can be seen from the figure that compared to partial duplication technique, the average area overhead of the proposed technique with dual rail implementation is always lower. Further area savings can be obtained if just a self-exercising checker is needed. For a target fault escape reduction of 95%, dual rail implementation is 25% better than partial duplication and the self-exercising implementation is 40% better.

It is to be noted that the area overhead numbers for partial duplication (PD in the table) are very low compared to the results presented in the original paper [5]. This difference can be attributed to the differences in selection methodology followed. In this paper, we performed fault simulations using traces from real programs instead of using random vectors at the inputs of combinational logic. We also considered only the faults which escape to the primary outputs

Table 4: % Area overhead for different target fault escape reductions for partial duplication (PD), proposed technique as a dual-rail checker (PT-D) and proposed technique as a self-exercising checker (PT-S)

	50% reduction			75% reduction			90% reduction			95% reduction			99% reduction		
	PD	PT-D	PT-S	PD	PT-D	PT-S	PD	PT-D	PT-S	PD	PT-D	PT-S	PD	PT-D	PT-S
module1	0.9	0.6	0.7	1.6	0.8	0.9	8.7	4.4	4.4	33.9	18	14.6	60.6	47.6	36
module2	0.9	0.6	0.7	8.3	8.4	7.8	51.2	38.8	31.4	67.5	64.2	48.5	90.1	106.2	77.5
module3	4.8	1.8	1.9	8.6	4.2	4.1	21.6	15.2	13	53.9	26.4	22.6	82.9	71.2	53.2
module4	12.3	9.2	10.1	15.9	14.8	15.7	25.5	20.8	20.7	27.1	25.4	24.9	30.9	31.8	31.1
module5	1.9	2.0	1.9	7.4	5.6	4.8	14.4	10.8	8.8	18.1	13.6	11.2	23.4	16.2	13.8

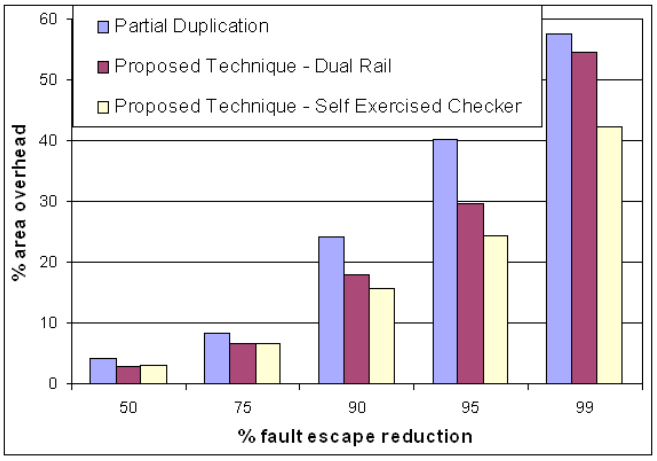

Figure 4: Average area overhead over different fault escape reductions

of the execution unit instead of considering all the faults which propagate to the outputs of the combinational logic.

6 Conclusions

A new algorithm for detecting transient faults in the control logic of a processor with a low overhead has been presented. An assertion checker is automatically constructed using the architectural traces of real programs. The checker checks the outputs of a combinational circuit against a subset of the truth table. The algorithm takes advantage of the following properties of the control logic of a processor to yield a low-overhead checker - asymmetry in the paths which are exercised at real-time and the asymmetry in the propagativity of bit-flips in individual flops to the architectural state of the processor. Fault simulation experiments were run on five different random control logic modules in an industrial processor. Results show that more than 95% of all the faults which propagate to architectural states can be detected with an average area overhead of just around 25%. This is more than 40% lesser when compared with previously proposed work for the same amount of fault detection.

Acknowledgements

We would like to acknowledge the contribution of Suriyaprakash Natarajan of Intel for early pattern-distribution results on a couple of Intel test cases demonstrating significant input vector bias.

References

[1] D. Das and N. A. Touba. Synthesis of circuits with low-cost concurrent error detection based on bose-lin codes. *J. Electron. Test.*, 15(1-2):145–155, 1999.

[2] P. Drineas and Y. Makris. Non-intrusive design of concurrently self-testable fsms. In *ATS '02: Proceedings of the 11th Asian Test Symposium*, pages 33–38, 2002.

[3] N. K. Jha and S.-J. Wang. Design and synthesis of self-checking vlsi circuits and systems. In *ICCD '91: Proceedings of the 1991 IEEE International Conference on Computer Design on VLSI in Computer & Processors*, pages 578–581, 1991.

[4] P. Liden, P. Dahlgren, R. Johansson, and J. Karlsson. On latching probability of particle induced transients in combinational networks. In *24th Int. Symposium on Fault-Tolerant Computing*, pages 340–349, 1994.

[5] K. Mohanram and N. Touba. Cost-effective approach for reducing soft error failure rate in logic circuits. In *Proceedings of the International Test Conference*, pages 893–901, 2003.

[6] H. T. Nguyen and Y. Yagil. A systematic approach to ser estimation and solutions. In *Proceedings of IEEE International Reliability Physics Symposium*, pages 60–70, 2003.

[7] M. Nicolaidis. Self-exercising checkers for unified built-in self-test (ubist). *IEEE Transactions on CAD*, 8(3):203–218, 1989.

[8] M. Nicolaidis, R. O. Duarte, S. Manich, and J. Figueras. Fault-secure parity prediction arithmetic operators. *IEEE Des. Test*, 14(2):60–71, 1997.

[9] P. Shivakumar et al. Modeling the effect of technology trends on the soft error rate of combinational logic. In *DSN '02: Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pages 389–398, 2002.

[10] S. S. Mukherjee et al. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 29, 2003.

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

[12] N. A. Touba and E. J. McCluskey. Logic synthesis of multilevel circuits with concurrent error detection. *IEEE Transactions on CAD*, 16(7):783–789, 1997.