Soft Error Detection via Double Execution with Hardware Assistance

Luis Bustamante and Hussain Al-Asaad
Department of Electrical and Computer Engineering
University of California — Davis
Davis, CA, USA
E-mail: {lbustamante, halasaad}@ucdavis

***Abstract*—As technology trends keep pushing cell dimensions in semiconductors to smaller geometries and higher densities, modern digital systems are increasingly becoming more vulnerable to reliability issues originated by soft errors. Various techniques used to detect soft errors are accomplished by incorporating redundancy into the hardware or software, but the penalty associated with the added redundancy can be measured by the high cost of the extra hardware or the degradation in performance for software-added redundancy. We are proposing a technique that compromise between hardware and software redundancy approaches. This approach is based on a software time redundancy combined with some hardware assistance. This hybrid technique has the potential to improve performance by adding a limited amount of hardware assistance when compared with a common time redundancy approach. It is also designed to set a foundation for further investigation into variations of this technique to improve soft error detection with better performance and less hardware.**

***Index Terms:* On-line testing, soft errors, double execution, error detection, and built-in self-test.**

I. Introduction

Soft Errors are caused by high-energy subatomic particles such as alpha particles and neutrons that may be originated from cosmic rays or radioactive decay. When these particles strike semiconductors, they create electron-hole pairs that temporarily produce charge variations that can flip the state of a transistor. Current trend projections show that the number of faults will continue to go up as we increase the transistor density [1]. Therefore, it is necessary to add protection measures to keep up with the increasing fault susceptibility trend.

Over the years, four main types of measures to protect against soft errors have evolved: circuit-level, logic-level, architectural-level, and software-level techniques [2].

- Circuit-level techniques include: (i) forward body bias—forward biasing the MOSFET body-source junction increases the junction capacitance, decreases the junction depletion charge collection volume, and creates a stronger feedback loop. All these factors decrease the probability that a particle strike can flip the output of the device. (ii) Transistor sizing—increasing the transistor aspect ratio increases output capacitance and hardens the transistor against charge injected by cosmic radiation; (iii) conservative design practices—using high-reliability components and excluding radiation-sensitive circuit styles such as dynamic-logic styles, and (iv) incorporating a sufficient functional margin in circuit designs to account for anticipated shifts in circuit characteristics.
- Logic-level techniques detect and recover from errors in combinational circuits by using a redundant or a self-checking circuit, such as output parity generation, to validate the output of the combinational circuit; and in flip-flop elements by providing redundant latches or by reusing scan flip-flops to hold redundant copies of flip-flop data.
- Architectural-level techniques include providing duplicate functional units or independent hardware to mimic and verify expected functionality and detect unexpected behavior.
- Software-level fault tolerance techniques such as replicating application execution through multiple communicating threads.

In the past, architects have often left reliability concerns to lower levels of the system stack. As the severity of these problems increases, however, low-level solutions will likely not suffice, and straightforward system level solutions such as blind redundancy will likely be too expensive for most market segments. Architects will therefore need to make reliability a first-class design constraint and develop new cost-effective approaches to reliability-aware design.

Most high-level soft-error detection approaches focus primarily on redundant execution [3]. Redundant execution requires running multiples replicas of a program and comparing the results. There are two main types of redundant execution techniques that are commonly used. The first one applies hardware redundancy and the second one uses what is known as software redundancy [4]. Redundant execution using hardware replicates hardware components to execute multiple copies of the program and then comparing the results. Two common hardware redundant execution techniques are known as Lock Stepping [3, 5] and Redundant Multithreading (RMT) [6, 7]. Both of these techniques require a significant amount of hardware redundancy which increases significantly design and silicon costs. For instance, Lock stepping is accomplished with a dual core processor where each core runs an identical copy of the program. Every cycle, the execution between the two cores

978-1-4673-0700-0/12/$31.00 ©2012 IEEE

is compared and a soft error is detected when a difference in between the 2 cores occur.

On the other hand, redundant execution through software redundancy is accomplished by using what is known as Time-redundancy. This technique relies on executing the same program in the same hardware twice and comparing results from both runs. One of the major advantages of using software redundancy to detect soft errors is that additional hardware is usually not necessary. The penalty incurred with software redundancy approach is the cost of the additional time required. There are different ways of accomplishing software redundancy but typically, it consists of replicating and running the instructions of a program and comparing their results [4]. A difference in the results of the two runs indicates that a soft error had occurred. The reason is that a fault caused by soft error is transitory and the probability of the same error occurring between the two executions is unlikely [3].

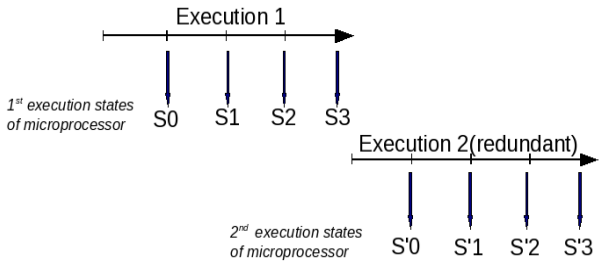

Figure 1. States S and S' generated during 1st and 2nd execution.

We are proposing a variation of the time-redundant execution method by incorporating a limited amount of hardware support. The function of the supporting hardware is to store the states (from the first execution) of the microprocessor into an external memory. As states are generated in the second execution (See Fig. 1), they are automatically compared to the stored states in the memory.

II. State Storage Module

A. State Storage Module SSM

The proposed method stores the state of the microprocessor on every clock cycle during the progression of the first code execution and then comparing the saved states with the states generated by the second execution of the same task. If during any cycle of second execution of the task a difference is detected in any of the microprocessor states, the program is stopped and the upper layer software is notified so that it can execute the appropriate recovery procedure.

If we assume that the microprocessor in Fig. 2 has n state registers, such that,

$$S_n = \{s_0, s_1, \ldots, s_{n-1}\} \quad (1)$$

Then the storage requirements to save the microprocessor states can be calculated by multiplying the number of register states n by the number of clock cycles m required to run the program or task at hand. Therefore, the hardware support storage requirements M is obtained as shown in Eq. 2.

$$M = n * m \quad (2)$$

All the states generated during the first execution are stored in memory with the assistance of the microprocessor state storage module SSM. When a particular task is targeted for error detection, the microprocessor signals the SSM to start storing the states of the microprocessor S_i by asserting "*execution_active*" and de-asserting "*second_execution*". The SSM will store the State information in SSM in the RAM bank in every clock cycle until the completion of the active task. The "*execution_active*" is deasserted at the end of the first task, the SSM stops storing the states of the microprocessor. At this point, all the states of the task have been stored in memory and the microprocessor is ready to start re-executing the same task a second time. The microprocessor accomplishes this by asserting simultaneously two signals: "*execution_active*" and "*second_execution*" to indicate to the SSM that the second execution of the task is in progress. The SSM starts retrieving the states S_i from the RAM bank and concurrently compares them with the states Si' being generated by the second execution. If any difference is detected by the logic in the SSM, then a signal E_i (shown in Eq. 3) is asserted to indicate that a fault has been detected.

$$E_i = ((S_0 \text{ xor } S_0')|((S_1 \text{ xor } S_1')|\ldots|(S_i \text{ xor } S_i')|\ldots \ldots|(S_{n-2} \text{ xor } S_{n-2}')|(S_{n-1} \text{ xor } S_{n-1}')) \quad (3)$$

An advantage of this error detection technique is that it can be used to identify the location of the fault and by knowing the clock cycle when the error is detected, it is possible to know the part of the code were the error was detected. While this information is not relevant for soft error detection, it can still be useful to identify potential hardware tolerance defects caused by manufacturing processes

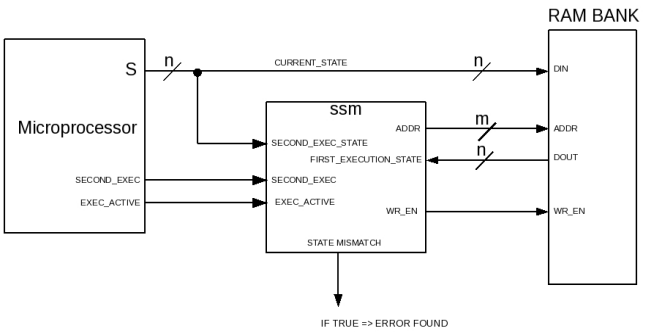

Figure 2. Hardware support to store/compare redundant execution states.

III. Implementation

A. Microprocessor

For the initial research, we implemented a custom Verilog model of a simple 32-bit microprocessor with floating point unit. To limit the complexity of the project, the microprocessor has a small instruction set and the soft error detection study was confined to the microprocessor's control unit (CU). The CU was chosen because of its interaction with every part of the microprocessor. This helped restrict the state space to 32 registers monitored by the SSM.

B. Extended Instruction Set

The microprocessor was modified to include 2 additional instructions needed to control the SSM. The first instruction added for this purpose is

FES RX, *Imm*

The FES instruction (First Execution Start) is designed to instruct the SSM to start saving the states of the microprocessor. The RX operand is the address in the register file that in turn contains the starting address in memory where the states of the microprocessor will be stored. The *Imm* operand is an immediate operand that indicates the period of the frequency in which the data is to be stored. If Imm=5'b00001 then the microprocessor SSM stores the states every clock cycle, if Imm=5'b00002 then the SSM is instructed to store the data every 2 clock cycles, etc.

The second instruction added is

SES RX, *Imm*

The SES instruction (Second Execution Start) instructs the SSM to retrieve and compare the data from the first execution already stored in memory with that of the second execution of the task in progress. The operands RX and *Imm* have the same function described by the FES instruction. The SES instruction must be followed by a Jump instruction that places the Program Counter at the beginning of the code task that will be run a second time.

IV. Results And Conclusions

The double execution via hardware assistance technique has been initially tested on a Fibonacci code program. The period of memory storage was limited to cycle per storage. The SSM successfully detected faults in the control unit.

The time from the occurrence of a soft error until it is detected is called the error latency. For full double execution, the maximum error latency can be $2*T$, where T is the execution time of the task. For our method, the maximum error latency is T. Hence we have a 50% reduction in the maximum error latency. The average error latency will be later computed via experimentation.

V. Future Research

A. Memory optimization.

The work presented in this paper is still in the early stages of research. Much more work still needs to be done, as this work is intended to serve as an initial step into other areas that require further investigation, such as:

1) Sample a subset of states every clock cycle: This will help identify critical states and to identify patterns to help understand if there are specific registers that provide more valuable coverage.

2) Sample all states at periodic intervals: To evaluate the effectivenss of sampling of certain intervals that might be sufficient to detect a soft error.

3) Sample a subset of the states at periodic intervals: This will incorporate the benefits of the above two areas.

B. Identification of Architectural Design patterns.

This might help identify and validate architectural structural advantages that can be incorporated into future designs of microprocessor that might be valuable on the detection of soft errors.

VI. References

1. A. Dixit, R. Heald, and A. Wood, "The Impact of New Technology on Soft Error Rates," *Proc. of IEEE Workshop on Silicon Errors in Logic-System Effects (SELSE 6*), Stanford, CA, March 2010.
2. R. K. Iyer, N.M. Nakka, Z. T. Kalbarczyk, and S. Mitra, "Recent Advances and New Avenues in Hardware-Level Reliability Support", *IEEE Micro*, pp. 18-29, November-December 2005.
3. S. Mukherjee, *Architecture Design for Soft Errors*, (Boston Morgan-Kaufmann), 2008.
4. M. Franklin, "A Study of Time Redundant Fault Tolerance Techniques for Superscalar Processors," *Proc. Defect and Fault Tolerance in VLSI Systems*, 1995, pp.207-215.
5. D. R. Siewiorek and R. S. Swarz, *Reliable Computer Systems Design and Evaluation*, A.K.Peters, Ltd., 1998.
6. S. S. Mukherjee, M. Kontz, and S. K. Reinhardt, "Detailed Design and Evaluation of Redundant Multithreading Alternatives", *Proc. International Symposium on Computer Architecture (ISCA)*, 2002.
7. M. Nicolaidis, "Time Redundancy-Based Soft-Error Tolerance to Rescue Nanometer Technologies", *Proc. IEEE VLSI Test Symp*., 1999, pp. 86-94.
8. N. Oh, P.O Shirvani, and E.J. McCluskey, "Error Detection by Duplicated Instructions in Super-Scalar Processors", *IEEE Transactions on Reliability*, Vol. 51, No. 1, pp. 63-75, March 2002.