SYSTEM-LEVEL EFFECTS OF SOFT ERRORS

A DISSERTATION

SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING

AND THE COMMITTEE ON GRADUATE STUDIES

OF STANFORD UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

Hyungmin Cho

August 2015

This dissertation is online at: http://purl.stanford.edu/zm955yw2192

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Subhasish Mitra, Primary Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

John Gill, III

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Oyekunle Olukotun

Approved for the Stanford University Committee on Graduate Studies.

Patricia J. Gumport, Vice Provost for Graduate Education

This signature page was generated electronically upon submission of this dissertation in electronic format. An original signed hard copy of the signature page is on file in University Archives.

Abstract

Radiation-induced transient errors (soft errors) are a major reliability concern for digital systems in advanced silicon CMOS technologies. Soft errors create unexpected changes in signal values during system operation, mostly in on-chip memories and flip-flops, resulting in undetected data corruption or expensive downtimes. This dissertation focuses on soft errors in flip-flops because design techniques to protect flip-flops are generally expensive. To protect on-chip memories, coding techniques are routinely used.

Error injection simulations are widely used for characterizing system-level effects of soft errors in a given design. These techniques generally inject single-bit errors into randomly-chosen locations (flip-flops, software-visible registers and memories) during randomly-chosen clock cycles. Flip-flop-level error injections suffer from slow Register-Transfer-Level (RTL) simulations. High-level error injections, that inject errors into software-visible registers or memories, are generally fast. Unfortunately, very little literature exists on the accuracies of high-level error injection techniques. We demonstrate that existing high-level error injections can be highly inaccurate by over an order of magnitude, and present detailed error propagation analysis to quantitatively explain the causes of such inaccuracies.

For fast, yet accurate, error injection simulations, we present a new mixed-mode simulation platform that combines simulators at two different abstraction levels. This platform achieves 20,000× speedup over RTL-only simulation for an industrial

multi-core chip consisting of approximately half-a-billion transistors. This platform targets soft errors in uncore components (e.g., memory subsystem, I/O controllers) that occupy significant portions of the overall chip area. Using this platform, we demonstrate, for the first time, that flip-flop soft errors in uncore components can significantly impact system-level reliability. We also demonstrate that recovery from uncore soft errors can be challenging for traditional system-level checkpointing techniques. A new replay technique overcomes these challenges for uncore components belonging to the memory subsystem.

Acknowledgments

First, I would like to express my sincere gratitude to my advisor, Prof. Subhasish Mitra, for his guidance, support, and encouragement. He inspired me with his tremendous commitment and enthusiasm for innovative research. During my degree progress, he always led me with visions and disciplines. I feel that I am very fortunate to have him as my academic advisor. Without him, my achievement would not have been possible.

I would like to thank my reading committee, Prof. Kunle Olukotun and Prof. John Gill. I also would like to thank my oral exam's chair, Prof. Phillip Wong. I also would like to extend my gratitude to Dr. Tanay Karnik, who kindly agreed to be on my oral's committee in spite of the inconvenience of the travel to Stanford.

I wish to thank all members of the Stanford Robust Systems Group for their enormous support and companionship. I would also like especially thank Dr. Young Moon Kim, Eric Cheng, David Lin, Thomas Shepherd, Dr. Mohamed Sabry, and Dr. Shahrzad Mirkhani with whom I have had close interactions. I have learned a lot from all of my fellow group members, and I will always treasure their friendship. Also, I thank Ms. Beverly Davis and Ms. Uma Mulukutla for being the best administrators and good friends.

I was fortunate to work with great collaborators and mentors including Prof. Jacob A. Abraham from the University of Texas at Austin, Dr. Chen-Yong Cher at IBM T. J. Watson Research Center, and many others. These wonderful collaborations have been one of the most important contributions for this dissertation.

Last but not least, I would like to thank my family and friends for their love and support. More than anyone else, I owe my deepest gratitude to my dear wife, Katherine Naeon Shin.

My studies at Stanford were supported by DARPA, FCRP GSRC, DTRA, STARnet SONIC, the National Science Foundation (NSF), Variability Expedition (NSF-sponsored), and Korea Foundation for Advanced Studies (KFAS).

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1. Soft Error Challenges

Radiation-induced transient errors (*soft errors*) are a major reliability concerns for digital systems in advanced technologies [Sanda 08, Seifert 12]. They cause transient errors, mostly in on-chip memories and sequential elements (latches and flip-flops). These errors can result in undetected data corruption or expensive downtimes.

To ensure robust operation of digital systems in the presence of soft errors, it is crucial to understand system-level effects of soft errors, i.e., how soft errors alter the behavior of the entire system and eventually affect the outcome of the application running on the system (*applicaton-level effects*). Proper understanding of such effects is important to achieve the desired level of system reliability in the presence of soft errors without incurring excessive costs; error protection techniques usually involve additional costs, such as power, performance, and area overheads [DeHon 10, Mitra 10].

In this dissertation, we study the accuracy levels associated with existing techniques for quantifying system-level effects of soft errors. Using extensive

simulation and emulation, we quantitatively compare accuracy levels and perform a thorough analysis on the sources of (in)accuracies associated with those mechanisms.

This dissertation also studies system-level effects of soft errors in uncore components[1], such as memory subsystem or I/O controllers. Uncore components account for a significant portion of today's large-scale systems (e.g., multi-core processors) [Li 13]. However, very few publications address soft errors in uncore components, unlike widely-studied soft errors in processor cores. Such limited understanding on soft errors in uncore components hinders efficient system design that are resilient to soft errors.

This dissertation focuses on radiation-induced soft errors in flip-flops (*flip-flop soft errors*) for several reasons:

1. Published data in literature, based on actual radiation test results, demonstrates that *flip-flop-level error injection*, which injects a single bit-flip to a randomly chosen flip-flop at a randomly chosen clock cycle, closely mimics system behavior in the presence of soft errors [Bottoni 14, Sanda 08].
2. Radiation testing results over several technologies (including 22nm FinFETs) show that combinational logic circuits are significantly less susceptible to soft errors [Seifert 12].
3. Design techniques to protect a system from flip-flop soft errors are generally expensive [Bernick 05, Borkar 07]. Coding techniques are routinely used for protecting on-chip memories [Kim 07, OpenSPARC].

[1] Also be referred to as "nest," "outside-core," or "northbridge". In this dissertation, we use this term to refer to components that are not processors or accelerators.

4. For permanent faults, there exist some published results comparing high-level fault models with gate-level stuck-at faults (mostly in the manufacturing testing literature and some in the architecture literature) [Maniatakos 11b]. In contrast, there exists very little analysis for temporary errors.

1.2. Contributions

1. We quantify the inaccuracies associated with various error injection techniques through detailed emulation and simulation results. Our results show high levels of inaccuracies (up to an order of magnitude) associated with widely-used soft error injection techniques compared to flip-flop-level soft error injection. We also explain the cause of such inaccuracies through a detailed analysis, which reveals that they directly model only a very small subset of system-level behaviors that can arise from flip-flop-level errors.
2. We present a simulation platform that is capable of simulating large-scale SoCs while modeling detailed flip-flop soft errors in uncore components. This approach overcomes the inaccuracies associated with existing soft error injection techniques while achieving high throughput for soft error simulations, when compared to soft error simulations that use the Register-Transfer-Level model of the entire system design (RTL-only simulation), this platform achieves over 20,000× speedup.
3. We present the first study of system-level effects of soft errors in uncore components in a large-scale industrial-grade OpenSPARC T2 SoC with 500 million transistors, eight processor cores, and many uncore components

[OpenSPARC]. We report quantified results on the effects of soft errors in L2 cache controllers, DRAM controllers, crossbar interconnects, and PCI Express I/O controllers. We show that soft errors in uncore components can have significant reliability impact comparable to that of processor cores.

4. Our soft error analysis shows that traditional system-level checkpoint recovery techniques that generally target processor cores are inadequate for uncore components since those techniques may incur long delays for system outputs. To overcome the challenge, we present a new soft error recovery technique called Quick Replay Recovery (QRR). We demonstrate the effectiveness of QRR for the L2 cache controller and the DRAM controller in the OpenSPARC T2 design. QRR results in 100× improvement (i.e., reduction) of the probability that an application run fails to produce correct results due to soft errors; the corresponding chip-level area and power impact for all L2C and MCU instances are 1.44% and 2.49%, respectively.

1.3. Outline

Chapter 2 discusses the inaccuracies associated with existing soft error injection techniques for simulation studies. In Chapter 3, we introduce a new simulation platform that accelerates soft error simulation for uncore components. Chapter 4 presents system-level effects of soft errors in uncore components and discusses soft error resilience challenges for uncore components. Finally, we conclude this dissertation in Chapter 5.

Chapter 2

Quantitative Evaluation of Soft Error Injection Techniques

2.1. Introduction

While radiation testing is generally successful in evaluating the soft error resilience of digital systems [Michalak 12, Sanda 08], simulation-based error injection techniques are also important at various stages of robust system design:

1. To analyze the application-level effects of soft errors.
2. To quantify the effectiveness of various soft error resilience techniques.
3. To make decisions about the set of error resilience techniques that must be used to protect a given design from soft errors.

Error injections at the flip-flop level can accurately capture the effects of flip-flop soft errors [Ramachandran 08, Sanda 08]. Such injections generally rely on slow

RTL simulations, sometimes with hardware acceleration or emulation. In contrast, error injections at higher abstraction layers are much less precise but can be very fast. Error injection techniques at high-level abstraction layers are important for understanding the application-level erroneous behaviors. The following abstraction layers are widely used (Table 2.1):

1. Software-level: Errors are often represented as single bit-flips in software variables, e.g., [Chen 08, Yim 10].
2. Architecture-level: Errors are injected into states defined by the Instruction Set Architecture (ISA). Single bit-flips in the architectural registers are often used, e.g., [Feng 10, Pattabiraman 11, Racunas 07, Zhang 10].
3. Micro-architecture-level: Error injection is performed using a detailed micro-architectural simulator. The internal states of the simulator are targets of error injection. Depending on the simulator implementation, such error injection targets may not always correspond to actual hardware components.

Table 2.1. Error injection techniques at various layers of abstraction.

Abstraction layer	Example platform	Performance (cycles / sec.)
Software	x86 processor [Yim 10]	3×10^9
Architecture	TSIM SPARC simulator [Leon]	6×10^7
Micro-architecture	gem5 simulator [Gem5]	3×10^6 (Simple CPU) 2×10^5 (Detailed CPU)
Flip-flop	IVM Alpha-like processor RTL simulation [Maniatakos 11b]	6×10^2
Flip-flop (Emulation)	OpenSPARC T1 FPGA emulation [Pellegrini 12]	10^7

To select a suitable error injection technique that meets the target accuracy and execution time requirements, one must address the following two essential aspects:

1. **Quantify** the inaccuracies of results obtained from error injection at various layers of abstraction.
2. **Analyze** the sources of these inaccuracies.

There exist very few publications that quantitatively address these questions. [Rimen 94] discusses inaccuracies of pin-level error injections that model only a small fraction (9-12%) of flip-flop errors. The authors also conclude that results from flip-flop error injections can match those from a special register-level error injection technique that injects register-level effects profiled from flip-flop error injections. However, the comparison is limited to a simple processor for which 80% of all flip-

flops belong to user-visible registers. [Rebaudengo 02] reports that register-file error injections can result in up to 400% inaccuracies for a version of the LEON processor [Leon]. However, the authors do not quantitatively analyze the sources of such inaccuracies.

[Arlat 03, Sanda 08] compare results obtained from actual error injection experiments with those obtained from error injection simulations / emulation. [Arlat 03] compares radiation, pin-level stuck-at faults, and electromagnetic interference experiments versus error injection simulations into program code and data. Although the authors report that error injections into program code and data can generate (erroneous) outcomes similar to actual error experiments, the observed rates of these outcomes can differ from actual error experiments. [Sanda 08] compares radiation tests versus flip-flop soft error injection, and concludes that results obtained from flip-flop soft error injections closely match those obtained from radiation experiments.

Some publications, e.g., [Kalbarczyk 99, Kanawati 93, Maniatakos 11b], profile high-level effects resulting from low-level errors, and use these high-level effects for quick error injection simulations. It has been pointed out in [Kanawati 93, Miskov-Zivanov 10, and numerous other papers on testing and high-level fault models] that a single flip-flop error can propagate through the system resulting in multiple error effects at the architecture- or software-level. However, there exists little work on systematic methodologies for deriving such high-level effects and for quantifying their effectiveness.

The lack of quantitative understanding of the accuracy trade-offs and their causes hinders progress in the evaluation and design of robust systems. This problem

becomes especially pronounced in the context of cross-layer resilience, where multiple error resilience techniques from different layers of the system stack cooperate to achieve cost-effective error resilience [DeHon 10, Mitra 10]. To achieve effective cross-layer resilience, error injection techniques must be able to capture low-level details accurately, simulate real-world applications in a scalable manner, and enable correct design decisions quickly.

In this chapter, we make the following contributions:

1. We quantify the inaccuracies of various error injection techniques through detailed FPGA-based emulation of the LEON3 in-order SPARC processor. Our results show high levels of inaccuracies (up to an order of magnitude) associated with high-level soft error injection techniques compared to flip-flop soft error injection.
2. In order to explain the sources of inaccuracies associated with high-level error injection techniques, we introduce a methodology which enables us to track, observe, and analyze how errors propagate through the system: from flip-flops all the way to the application outputs.
3. We explain why high-level error injection techniques directly model only a very small subset of system-level behaviors that can arise from flip-flop-level errors.
4. We demonstrate the generality of our results through RTL simulations of a super-scalar and out-of-order processor [Wang 04].

2.2. Error Injection Methodology

We created an FPGA-based error injection system by mapping a LEON3 processor (in-order and SPARC-based [Leon]) on the BEE3 emulation system [Davis 09] using Xilinx Virtex-5 FPGAs (Fig. 2.1). The LEON3 processor is a good choice for experimentation because the entire system, including L1 and L2 caches and the DRAM controller, can be mapped on the emulation platform. As a result, a large number of error injections can be performed. Moreover, in-order processor cores are often used in multi- and many-core SoCs [Borkar 11, Howard 10, OpenSPARC]. We designed the system with appropriate hardware support to track the propagation of injected errors through various layers of the system stack (details in Sec. 2.5.1). Such a setup enables us to compare various error injection techniques and analyze their inaccuracies using the same consistent environment for the same set of applications. In order to minimize the sensitivity of our results to the LEON3 architecture, we also present results using another set of error injections (fewer than LEON3) through RTL simulations of the IVM processor, which is a super-scalar and out-of-order processor [Wang 04].

Figure 2.1. BEE3 emulation system using Virtex-5 FPGAs.

2.2.1. Benchmark Applications

We used 11 out of 12 applications from the widely-used SPECINT 2000 benchmark suite [1]. We used the MinneSPEC workload for the input dataset [KleinOsowski 02]. The execution times of the benchmark applications range from 1.4×10^8 cycles to 3.5×10^9 cycles, and the cycle-per-instruction (CPI) values range from 1.54 to 3.36 (Table 2.2).

[1] We excluded the perlbmk application because it requires extensive file system support that was not modeled in our emulation system.

Table 2.2. SPECINT 2000 benchmark applications.

Name	Execution time (cycles)	CPI
bzip2	2,429M	1.97
crafty	294M	2.43
eon	3,479M	1.54
gap	145M	2.29
gcc	216M	2.71
gzip	2,753M	2.25
mcf	627M	3.37
parser	683M	1.84
twolf	751M	1.66
vortex	222M	2.60
vpr	424M	1.66

2.2.2. Error Injection Samples

It is impossible to inject all possible error scenarios (injection target × execution cycle). Hence, for each error injection technique and each application, we collected results from 40,000 or more error injection runs. (For error propagation and tracking analysis in Sec. 2.5, we report results from 320,000 flip-flop error injection runs for the LEON3 processor emulation, and from 160,000 flip-flop error injection runs for the IVM processor simulation.) Our error injection system initializes all the system states to their default reset values before each error injection run. A state is set to zero if its reset value is undefined, e.g., cache arrays and DRAM.

We implicitly assume that each flip-flop has the same (raw) soft error rate, similar to [Ramachandran 08, Seifert 10, Wang 04, 07], for the ease of reporting

results. For situations where this assumption may not apply, the observed results from error injections may change. However, the error injection methodology used in this dissertation still can be applied for those situations. To determine the confidence intervals of the error injection outcomes, we use a derivation similar to [Choi 90] and many other publications. With a sample size of 40,000 error injections, the 95% confidence interval is smaller than ±0.1% when the observed outcome rate is 1%. This derivation implicitly assumes that the system behavior with respect to soft errors is statistically similar during the execution, i.e., the probability of a certain outcome does not change according to a particular phase of the execution.

2.2.3. Error Injection Techniques

We compare five error injection techniques across three abstraction layers. To ensure proper initialization, no error is injected during the first 10,000 clock cycles (warm-up period).

1. **Flip-flop:** For each error injection experiment, the content of a randomly-chosen flip-flop is flipped during a randomly-chosen clock cycle. SRAM structures, e.g., register-file and cache, are not included because they are generally protected using ECC and parity[2]. Each flip-flop in the processor is an error injection target (1,250 flip-flops for LEON3, 13,877 flip-flops for IVM). The results obtained from flip-flop error injections are treated as "ground truth" because they closely mimic actual soft error effects [Sanda 08].

[2] Error injection results that include register-file errors can be derived by combining flip-flop error injection results and RegU error injection results (defined in 2.3.2a).

2. Register-file: Error injection into software-visible registers is widely used. Depending on the target system architecture and simulator capabilities, various error injection studies use slightly different register-file error injection techniques.

   2a. **Register Uniform (RegU):** For each error injection experiment, a single-bit error is injected into a randomly-chosen bit location of a randomly-chosen register during a randomly-chosen clock cycle. The target register set includes all general-purpose registers, stack pointer, and branch pointer. This type of error injection is used in [Feng 10, Zhang 10]

   2b. **Register Write (RegW):** For each error injection experiment, during a randomly-chosen clock cycle, a single-bit error is injected into a randomly-chosen bit location of a register being written into during that clock cycle. If no register is being written into during that clock cycle, the error injector waits for the next instruction that writes into a register, and injects error into a randomly-chosen bit location of that register. The target register set is the same as that for RegU. This type of error injection is used in [Pattabiraman 11, Racunas 07].

3. Program Variable: Errors are injected into application software (program) variables: global data, heap, and stack.

   3a. **Program Variable Uniform (VarU):** For each error injection experiment, a single-bit error is injected into a randomly-chosen bit location of a randomly-chosen program variable during a randomly-chosen clock cycle. The target program variables include all variables in memory (stack, global

data, and heap) at the chosen error injection cycle; i.e., the target set includes memory locations actually used by the program. No error injection is performed into freed heap objects or intermediate variables eliminated during compilation. This type of error injection is used in [Chen 08, Yim 10].

3b. **Program Variable Write (VarW):** For each error injection experiment, during a randomly-chosen clock cycle, a single-bit error is injected into a randomly-chosen bit location of a program variable being written into during that clock cycle. If no program variable is being written into during that clock cycle, the error injector waits for the next instruction that writes into a program variable, and injects an error into a randomly-chosen bit location of that variable. The target variable set is the same as that for VarU. A similar error injection technique is used in [Chen 06, Gu 04].

2.3. Results: Error Injection Inaccuracies for the LEON3 processor

The *outcome* of an error injection run can be categorized into one of the following categories [Sanda 08, Wang 04, 07]:

1. **Vanished**: The application terminates normally, and at the end of the execution, the output files and all architectural states match with those obtained from the error-free run.
2. **Application Output Not Affected (ONA):** The application terminates normally without any error indication, and, at the end of the execution, the output files from the erroneous run match those obtained from the error-free run. However,

one or more remaining bits of the architectural state differ from those obtained from the error-free run.

3. **Application Output Mismatch (OMM):** The application terminates normally without any error indication. However, at the end of the execution, the output files of the application are different from those obtained from the error-free run. The remaining architectural state bits may or may not match with those of the error-free run. This category is often referred to as silent data corruption (SDC) as well [Sanda 08, Michalak 12].
4. **Unexpected Termination (UT)**: The application terminates abnormally with error indication. These include error reporting interrupts, e.g., divide-by-zero, invalid instruction, or memory access violation, and application-detected errors, e.g., exit() function calls with error codes.
5. **Hang**: The application does not produce any result or does not terminate within a specified timeout limit set to 2× the nominal execution time[3].

Figure 2.2 compares the observed rate of each outcome category (*outcome rate*) obtained from 40,000 error injections for each of the 11 applications using each of the 5 error injection techniques. The 95% confidence intervals from the error injection samples are shown as the error bars. (We verified the integrity of our error injection results by comparing results obtained from 10 subsamples, 4,000 error injection runs each, for each error injection technique. The differences in the outcome rates across these subsamples are less than 2%.)

[3] The nominal execution time is measured on warmed-up caches (without error injection).

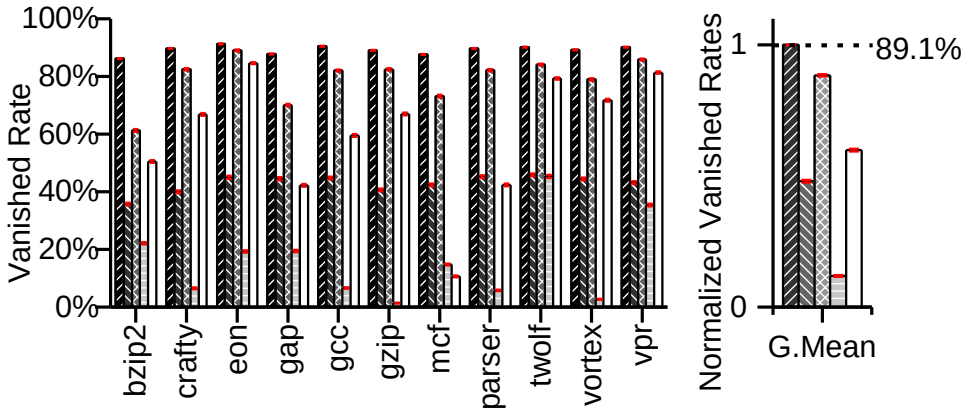
To compare the results for various injection techniques, consider the OMM rate of the *crafty* application in Fig. 2.2c as an example. 1.3% of flip-flop error injections result in OMM. However, high-level error injections result in different rates for the same outcome: 2.7% for RegU, 6.7% for RegW, 0.53% for VarU, and 16.1% for VarW. These differences are well beyond the confidence intervals (e.g., the 95% confidence interval for the OMM rate of the *crafty* application is ±0.11% for flip-flop error injection and ±0.24% for RegW error injection).

Since outcome rates vary across applications even for the same error injection technique (e.g., the OMM rate obtained from flip-flop error injection varies from 0.02% for the *parser* application to 1.78% for the *twolf* application), the inaccuracy levels are compared using *normalized outcome rates* with respect to the corresponding flip-flop error injection results. For example, for the *parser* application, the 0.04% OMM rate obtained from RegU error injection is 2× that of the corresponding flip-flop error injection result. Figure 2.2 also compares these normalized outcome rates using geometric means according to the following expression [Fleming 86]:

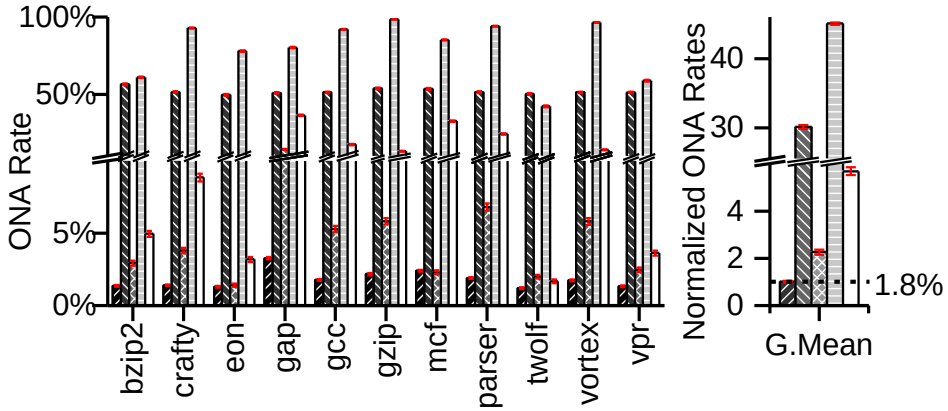
$$\text{G.Mean}(x,t) = \sqrt[n]{\prod_{i=1}^{n} \frac{\text{Rate}(x,t,i)}{\text{Rate}(x,\text{flip-flop},i)}} \quad (2.1)$$

where x is an outcome, t is an error injection technique, and n is the number of benchmark applications. $Rate(x,t,i)$ is the observed rate of outcome x when error injection technique t is applied for application i .

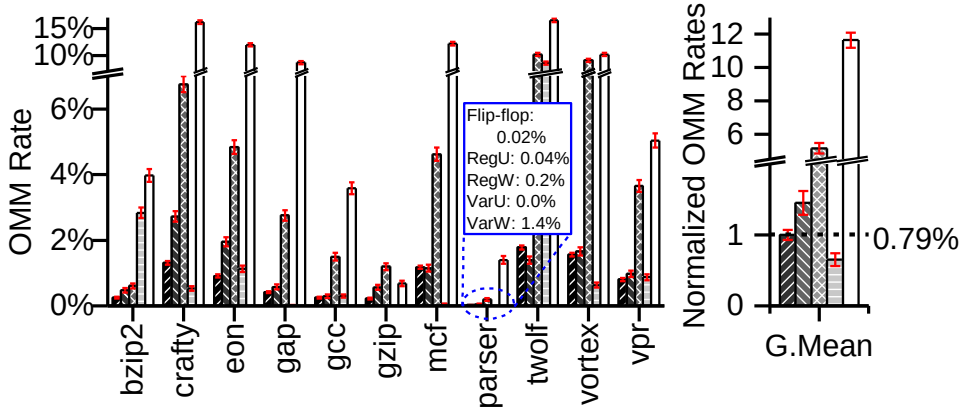
While geometric means show overall inaccuracy levels, they may not capture how inaccuracy levels vary across various applications. For example, consider the UT outcome type in Fig. 2.2d. The geometric mean of normalized UT outcome rates for RegW error injection is 1.15× that of flip-flop error injection. However, the normalized UT outcome rates for RegW error injection (with respect to flip-flop error injection) vary from 0.5× to 3× across applications.


100%
80%
60%
40%
20%
0%
Vanished Rate
bzip2
crafty
eon
gap
gcc
gzip
mcf
parser
twolf
vortex
vpr
1
0
89.1%
Normalized Vanished Rates
G.Mean

(a)


100%
50%
5%
0%
ONA Rate
bzip2
crafty
eon
gap
gcc
gzip
mcf
parser
twolf
vortex
vpr
40
30
4
2
0
1.8%
Normalized ONA Rates
G.Mean

(b)


15%
10%
6%
4%
2%
0%
OMM Rate
Flip-flop:
0.02%
RegU: 0.04%
RegW: 0.2%
VarU: 0.0%
VarW: 1.4%
bzip2
crafty
eon
gap
gcc
gzip
mcf
parser
twolf
vortex
vpr
12
10
8
6
1
0
0.79%
Normalized OMM Rates
G.Mean

(c)

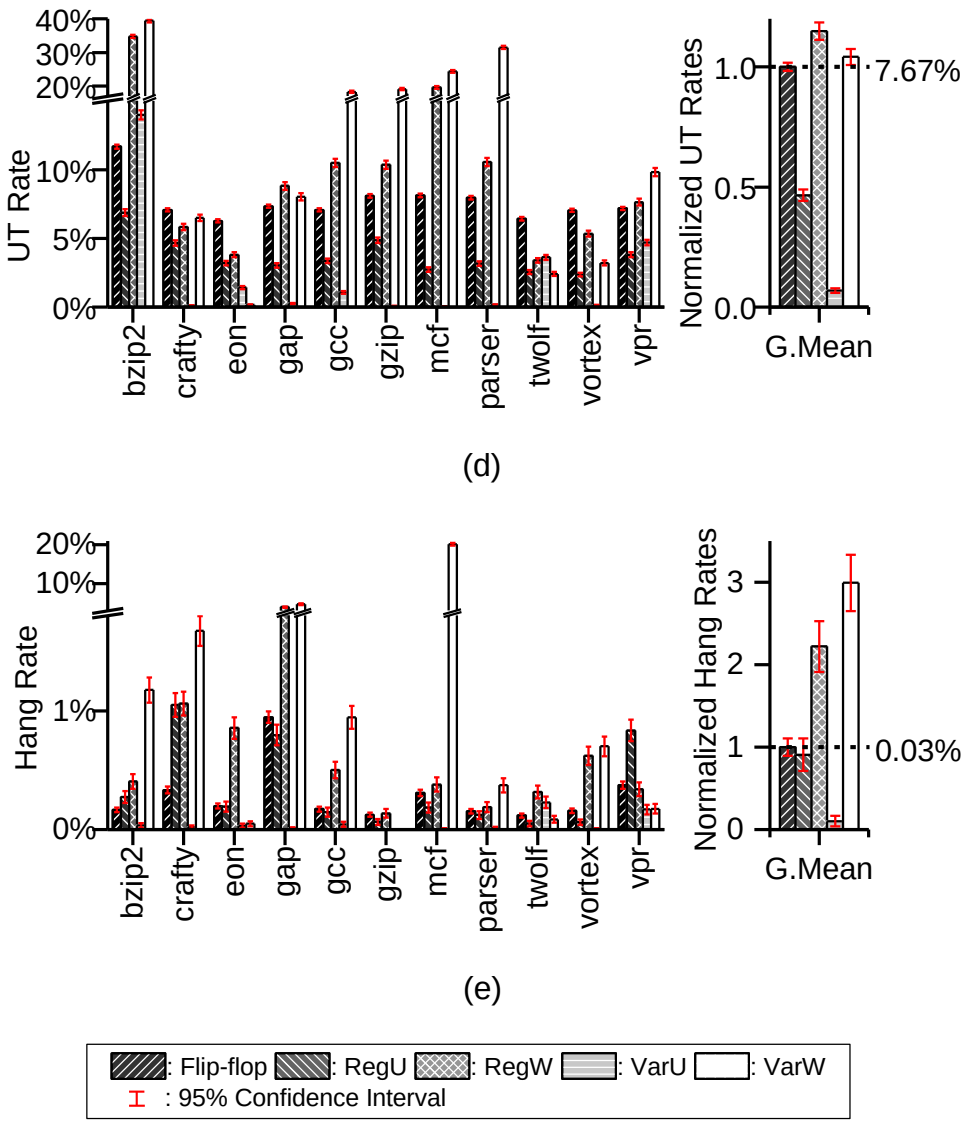


Figure 2.2. Comparison of the observed rate of each outcome type obtained from various error injection techniques for the LEON3 processor. (a) Vanished. (b) ONA. (c) OMM. (d) UT. (e) Hang.

Key observations from the above results are:

1. Existing high-level error injection techniques, **that inject single errors into registers or program variables**, can result in high degrees of inaccuracies by more than an order of magnitude. For example, the geometric means of normalized outcome rates obtained from VarU error injection can range from 0.07× (for the UT outcome type) to 45× (for the ONA outcome type) when compared to the corresponding flip-flop error injection results.
2. There is no single trend (e.g., always optimistic or always pessimistic) that can explain the inaccuracies associated with existing high-level injection techniques. For example, the studied high-level error injection techniques generally tend to overestimate OMM outcome rates, but the RegU and VarU error injection techniques tend to underestimate UT outcome rates.
3. RegU and VarU error injection techniques result in very high degrees of ONA outcome rates. These techniques select error injection targets uniformly over the entire target space, and, hence, may inject errors into locations that may not be accessed during the rest of the application execution (resulting in ONA outcomes at the end of the execution). These situations can potentially contribute to the very high ONA outcome rates for RegU and VarU error injection techniques.

2.4. Results: Error Injection Inaccuracies for the IVM processor

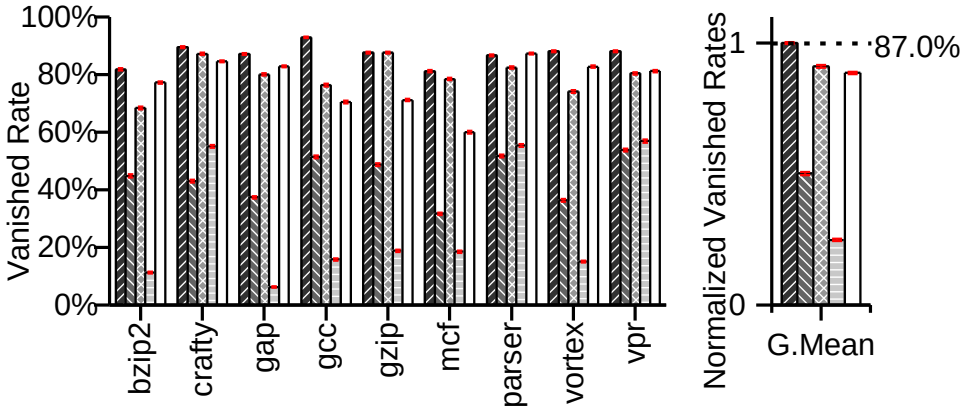
Similar to Sec. 2.3, we present a comparison of inaccuracies associated with various error injection techniques for the IVM processor[4]. We use the same benchmark applications[5] and error injection techniques as for the LEON3 error injections. Figure 2.3 compares the observed rate of each outcome obtained from 40,000 error injections for each of the nine applications using each of the five error injection techniques. The results show trends similar to the LEON3 error injections.

1. Existing high-level error injection techniques, **that inject single errors into registers or program variables**, can result in high degrees of inaccuracies by more than an order of magnitude. For example, the geometric means of normalized outcome rates obtained from VarU error injection can range from 0.07× (for the Hang outcome type) to 22× (for the ONA outcome type) when compared to the corresponding flip-flop error injection results.
2. There is no single trend (e.g., always optimistic or always pessimistic) that can explain the inaccuracies associated with existing high-level error injection techniques. For example, the RegU error injection technique generally tends to underestimate Hang outcome rates, but it tends to overestimate OMM outcome rates. Also, the RegU error injection technique generally tends to underestimate UT outcome rates while RegW error injection technique tends to result in overestimated UT outcome rates.

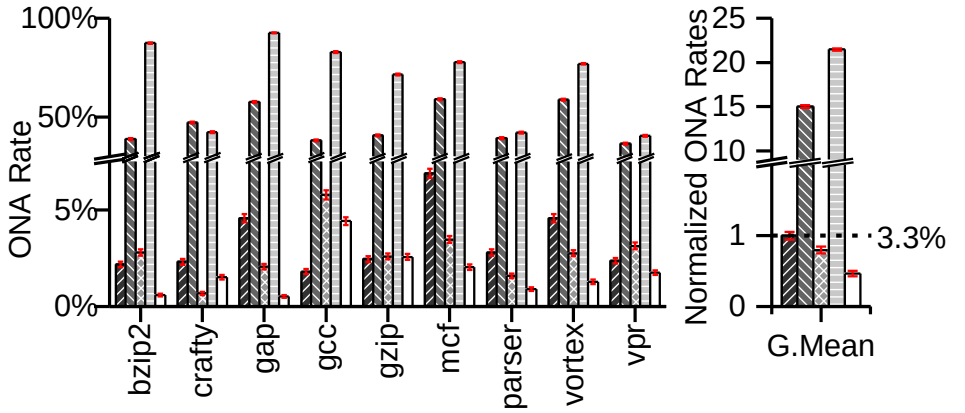
[4] Error injection simulations for IVM processor were conducted by Dr. S. Mirkhani and Prof. J. A. Abraham at The University of Texas at Austin. The simulations were performed using the Stampede supercomputer at the Texas Advanced Computing Center.

[5] We excluded *eon* and *twolf* since these applications very frequently use floating point instructions which are not supported by the existing IVM processor model.

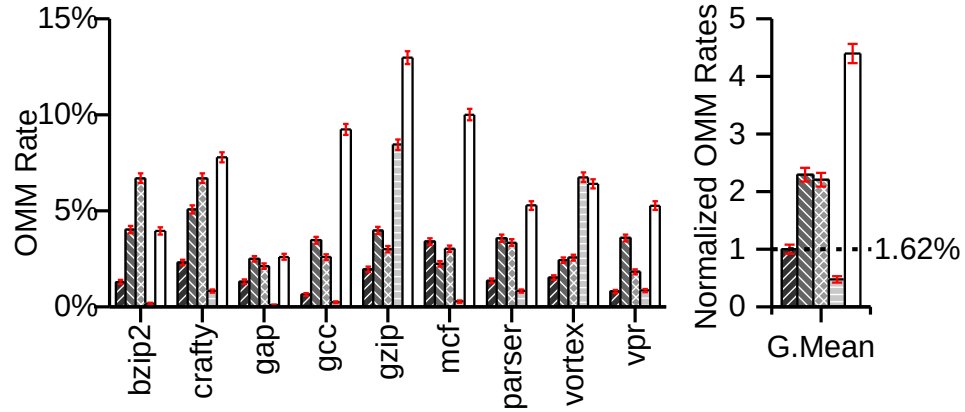
3. RegU and VarU error injection techniques result in very high ONA outcome rates.



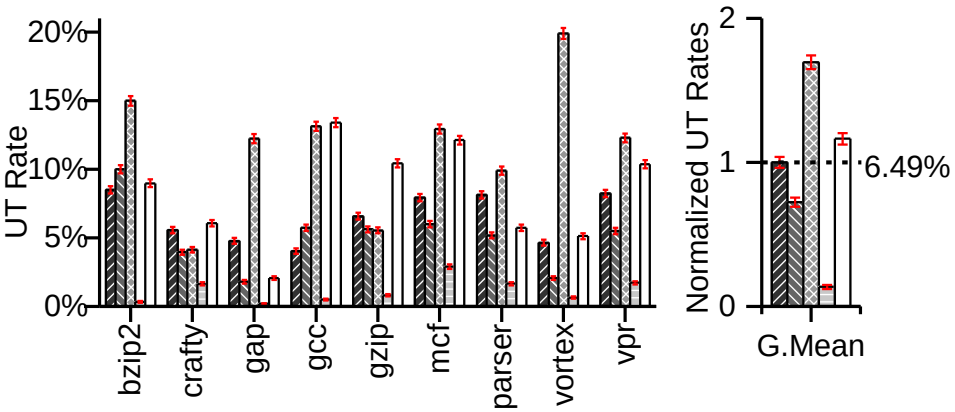
(a)



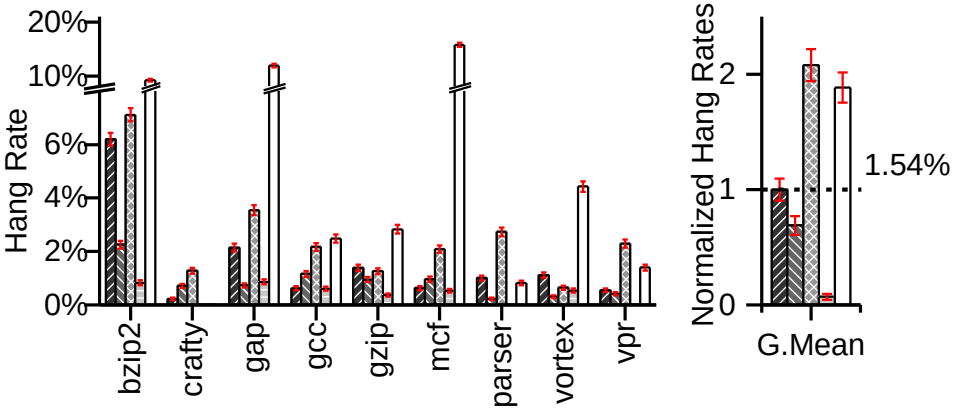
(b)



(c)



(d)



(e)

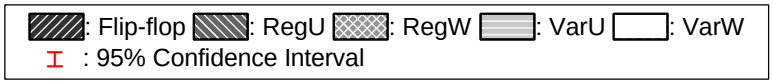


Figure 2.3. Comparison of the observed rate of each outcome type obtained from various error injection techniques for the IVM processor. (a) Vanished. (b) ONA. (c) OMM. (d) UT. (e) Hang.

2.5. Results: Inaccuracy Analysis

To understand the causes of inaccuracies associated with high-level error injection techniques (Sec. 2.3 and 2.4), we track how flip-flop errors propagate and affect high-level system states. This is enabled by special hardware support that we implemented in our FPGA-based emulation system for the LEON3 processor (details in Sec 2.5.1). The inaccuracies generally result from the following limitations of existing high-level error models:

1. They generally inject a single error into a single location, e.g., a single register or a single program variable, during a (randomly-chosen) clock cycle.
2. Error injection targets do not always cover all software-visible states. For example, error injection targets are often limited to general-purpose registers while flip-flop errors can propagate to other software-visible states, e.g., integer condition codes and interrupt modes for the LEON3 processor (details in Table 2.6).

2.5.1. Flip-flop Error Propagation Tracking

In the FPGA-based emulation system for the LEON3 processor and the simulation environment for the IVM processor, we implemented two copies of the processor cores, the *erroneous core* and the *golden core* (Fig. 2.4). Flip-flop errors are injected only into the erroneous core. The golden core shares the same general-purpose registers, main memory (L1 and L2 caches, off-FPGA DRAM). All the other modules on the emulated system, e.g., the interrupt controller and the Ethernet controller, are shared between the two cores. These modules have memory-mapped

interfaces and are accessed through load and store operations only. The address space for these modules is not accessible by user applications, and any access to these modules triggers memory access violations resulting in UT outcome.

We implemented a set of *propagation checkers* to detect if a flip-flop error in the erroneous core results in a mismatch with the golden core (i.e., *error propagation* occurs). The *R-checker* (*Register-checker*) and the *M-checker* (*Memory-checker*) detect error propagation to the general-purpose registers (hereinafter referred to as "register-file") and the main memory (hereinafter referred to as "memory"), respectively. The address inputs, the data inputs, and the read and write enable signals to the register-file from both cores are compared by the R-checker at every cycle to detect if there is any mismatch. Since load and store operations access program variables through the memory interface, the M-checker compares the address inputs, the data inputs, and the read and write enable signals to the memory from both cores to track error propagations to program variables in the memory.

The *I-checker* (*Instruction-checker*) compares the instruction fetch addresses to detect if the erroneous core fetches instructions from incorrect memory locations. Executing instructions fetched from incorrect memory addresses can eventually result in mismatches in the register-file or memory. Depending on how one implements high-level error injection (e.g., error injection techniques that target program variables only vs. error injection techniques target the entire memory space including instructions), the effects of the errors detected by the I-checker may or may not be captured.

Suppose that, upon injection of a flip-flop error during clock cycle *i*, a mismatch is detected by the R-checker during clock cycle *j* (and no more mismatch is detected by any other checker). This detection indicates the propagation of the injected flip-flop error to the register-file. If no more mismatches are detected until the end of program execution, the effect of the injected flip-flop error during clock cycle *i* is identical to the injection of the detected register error(s) during clock cycle *j*.

A mismatch detected by the R-, M-, or I-checkers (*initial propagation*) may cause other registers or memory locations to have incorrect values (*subsequent propagation*). For example, a flip-flop soft error may corrupt the value of register *A* (initial propagation; detected by the R-checker). If the processor core updates register *B* with the results of an operation involving the corrupted value in register *A*, register *B* may also have an incorrect value (subsequent propagation). To identify if the injected flip-flop error can be modeled by high-level error injection techniques, our error propagation tracking mechanism detects initial propagations only. Otherwise, a flip-flop error that could be modeled by a single error injection into the register-file or memory (e.g., a flip-flop error that results in a single initial propagation during the execution followed by multiple subsequent propagations) might be incorrectly classified as a flip-flop error that would require multiple error injections into the register-file or memory. In the previous example, if the R-checker detects both initial and subsequent propagations, the checker triggers multiple times (propagations to registers *A* and *B*); in reality, the flip-flop error can be modeled by injecting only a single error corresponding to the initial propagation to register *A* only.

In our error propagation tracking mechanism, the golden core and the erroneous core share the same (erroneous) register-file and memory. Therefore, these two cores have the same register-file and memory state after the detection of the initial propagation, and both cores behave in the same way during subsequent propagation (i.e., R- and M-checkers do not trigger during subsequent error propagation).

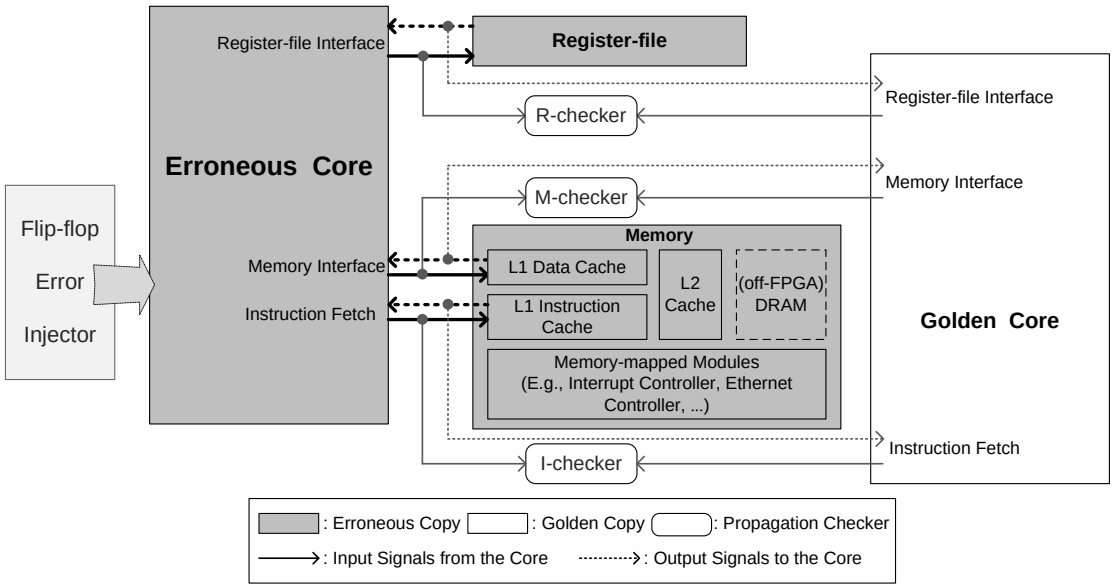


Figure 2.4. Error propagation detection logic.

2.5.2. Flip-flop Error Propagation Types

The effect of an injected flip-flop error can be categorized into one of the following types (Fig. 2.5) depending on its propagation **during** (not necessarily at the end of) program execution.

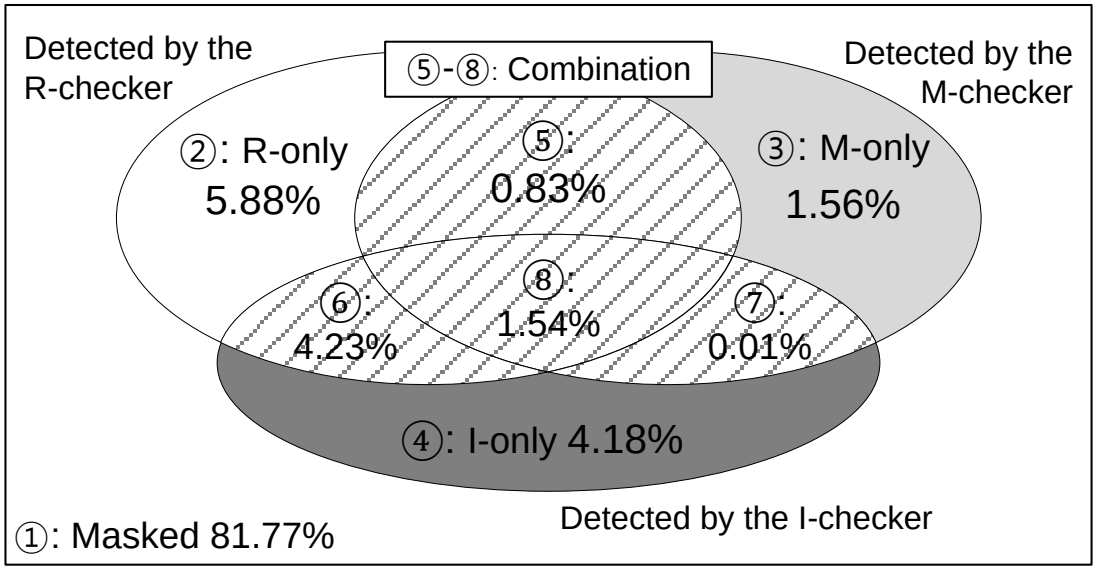
1. **Masked**: The flip-flop error **never** propagates to the register-file or memory at any point during the execution, i.e., no propagation checker in Fig. 2.4 detects a

mismatch. Since no register or program variable has a different value compared to the error-free golden execution at any point during the execution, a masked flip-flop error results in the Vanished outcome only. The converse is not necessarily true.

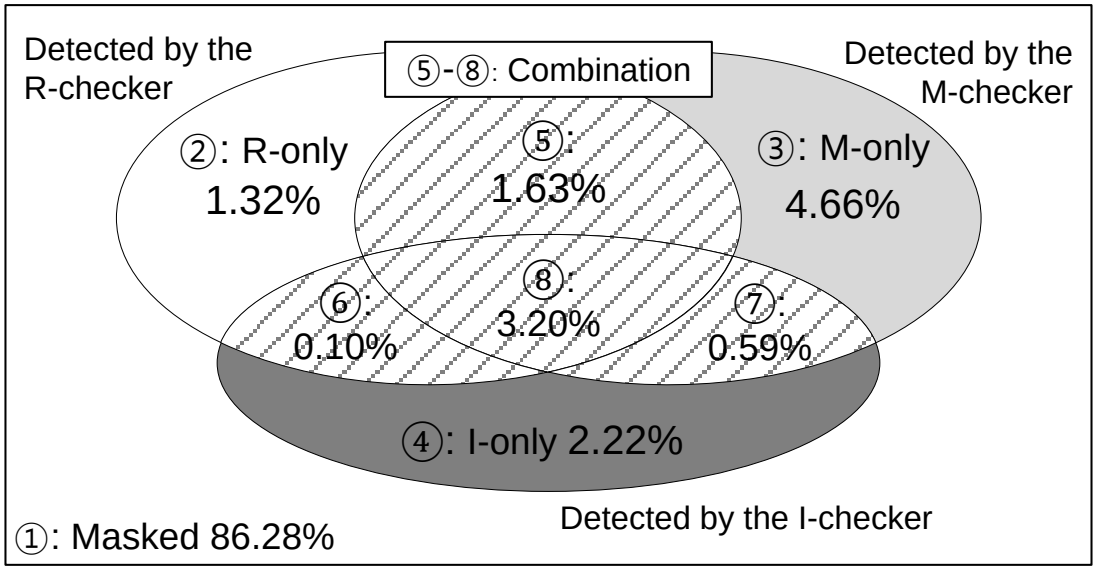
2. **Register-only (R-only)**: The flip-flop error propagates only to the register-file and corrupts register-file contents, i.e., only the R-checker triggers (once or multiple times) during program execution. For example, if a flip-flop error injected during cycle *i* directly propagates to a register during cycle *j*, the R-checker triggers during cycle *j*. If the erroneous flip-flop also corrupts additional flip-flops, and those additional flip-flop errors propagate to the register-file during clock cycles *k, l,…*, the R-checker can trigger multiple times during the execution. A mismatch in the read or write enable signal or the address input is categorized as an R-only propagation type because it can be modeled as single or multiple register-file error injections.
3. **Memory-only (M-only)**: The flip-flop error propagates only to the memory and corrupts program variable contents, i.e., only the M-checker triggers (once or multiple times) during program execution.
4. **Instruction-only (I-only)**: The flip-flop error affects the instruction fetch addresses, i.e., only the I-checker triggers (once or multiple times) during program execution. Due to the instructions fetched from incorrect addresses, the application may eventually result in mismatches in the register-file or memory (when compared to the error-free execution); these subsequent error propagations do not trigger the R- or M-checkers.

5. **Combination**: The flip-flop error results in error propagations that cause two or more of the R-, M-, and I-checkers to trigger. As discussed before, errors detected by one of the checkers do not result from the propagation of errors detected by the other checkers.

Figures 2.5 shows a Venn diagram with the percentages of various propagation types. For the LEON3 processor, the figure shows percentages resulting from 3,520,000 flip-flop error injections (320,000 flip-flop error injections for each of 11 benchmark applications). For the IVM processor, the numbers are obtained from 1,440,000 flip-flop error injections (160,000 flip-flop error injections for each of 9 benchmark applications). High-level error injection techniques that inject errors either into the register-file or memory target error propagation types ② or ③ (e.g., 7.44% of injected flip-flop errors or 40.81% of non-masked flip-flop errors for the LEON3 processor). However, high-level error injection techniques not always model the effects of all register-file or memory propagations. Many existing high-level error injection techniques 1) inject single bit-flip errors 2) into a single target (single register or single memory word) 3) at a single cycle (as discussed in Sec. 2.2.3). In Sec. 2.5.4, we show that only small portions of regieter-file and memory propagations are ***directly*** modeled by single-bit error injections into the register-file or memory.



(a)



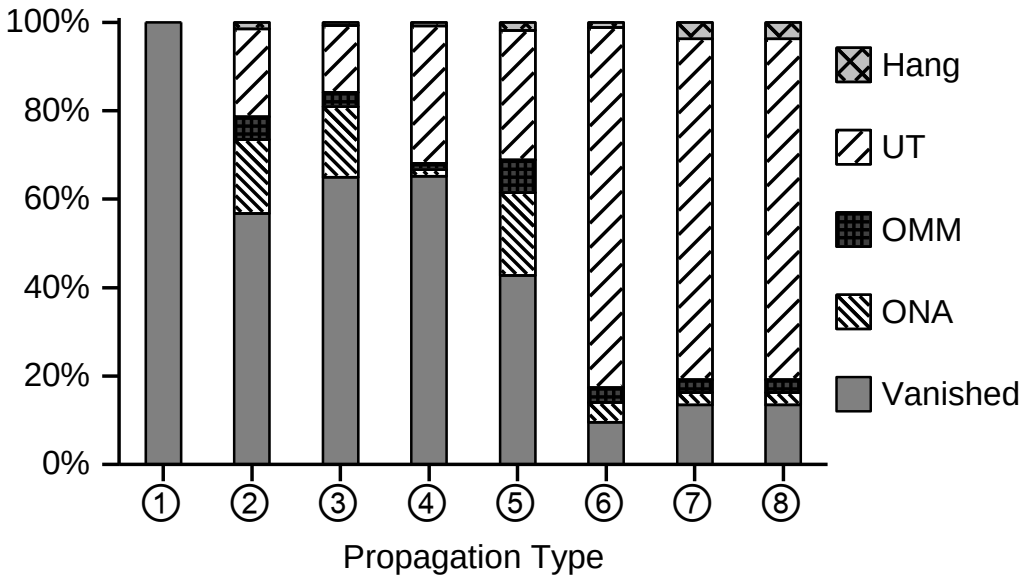
(b)

Figure 2.5. Flip-flop error propagation types and their observed rates. (a) Flip-flop error injections into the LEON3 processor. (b) Flip-flop error injections into the IVM processor.

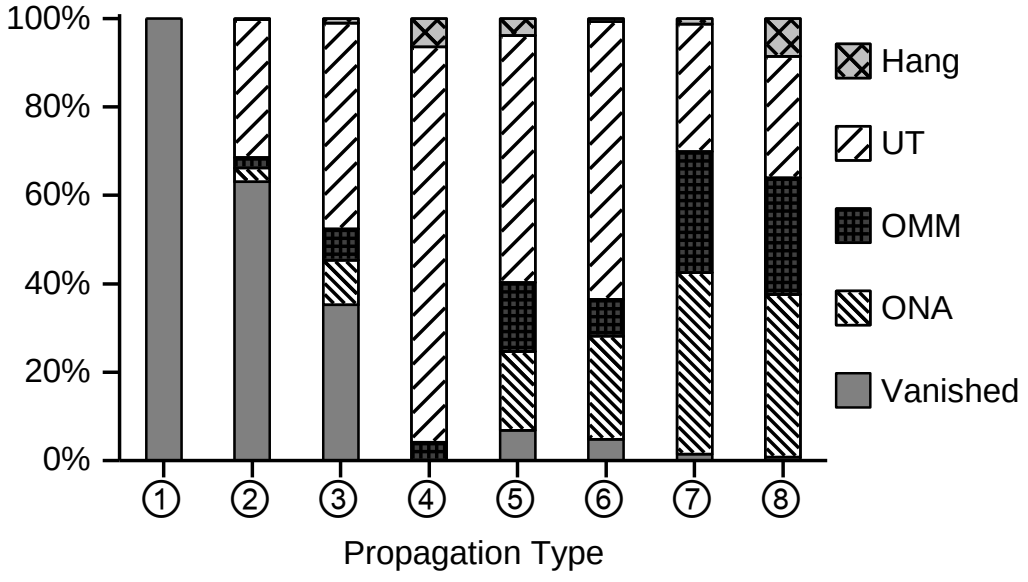
2.5.3. Outcomes versus Flip-flop Error Propagation Types

Flip-flop errors resulting in various error propagation types (Sec. 2.5.2) ultimately produce various outcome rates at the end of program execution. In Fig. 2.6, for each propagation type, we report its percentage that results in a given outcome. For example, 100% of flip-flop error injections that result in masked propagation type (①) produce the Vanished outcome type only because those flip-flop errors does not have any effects on register values or program variable values during the execution. Flip-flop soft errors that result in error propagations detected by all propagation checkers (propagation type ⑧) frequently result in the UT type outcomes (77.2% for the LEON3 processor).

In Tables 2.3 and 2.4, for each outcome type, we report the percentage of that outcome contributed by each propagation type. For the Vanished outcome, the masked propagation type (①) is the dominant cause, but other propagation types also contribute as well. For the remaining outcomes, correct modeling of all non-masked error propagation types (②-⑧) is crucial.



(a)



(b)

Figure 2.6. Percentage of a propagation type resulting in an outcome. (a) Flip-flop error injections into the LEON3 processor. (b) Flip-flop error injections into the IVM processor.

Table 2.3. Percentage of an outcome contributed by a propagation type for the LEON3 processor.

Outcome / Propagation Type	Vanished	ONA	OMM	UT	Hang
Masked (①)	92.54%	0.00%	0.00%	0.00%	0.00%
R-only (②)	3.11%	57.26%	46.26%	15.22%	33.68%
M-only (③)	0.92%	15.86%	6.94%	3.12%	4.24%
I-only (④)	2.53%	3.51%	9.13%	17.23%	12.87%
Combination (⑤)	0.33%	9.17%	8.24%	3.21%	7.78%
Combination (⑥)	0.37%	11.49%	22.54%	45.47%	19.21%
Combination (⑦)	0.01%	0.02%	0.11%	0.04%	0.15%
Combination (⑧)	0.19%	2.69%	6.78%	15.71%	22.07%
Total (①-⑧)	100.00%	100.00%	100.00%	100.00%	100.00%

Table 2.4. Percentage of an outcome contributed by a propagation type for the IVM processor.

Propagation Type \ Outcome	Vanished	ONA	OMM	UT	Hang
Masked (①)	97.05%	0.00%	0.00%	0.00%	0.00%
R-only (②)	0.94%	1.89%	1.79%	6.24%	0.69%
M-only (③)	1.85%	20.88%	19.23%	33.00%	8.53%
I-only (④)	0.00%	0.03%	5.23%	30.04%	26.54%
Combination (⑤)	0.13%	12.96%	14.90%	13.85%	11.61%
Combination (⑥)	0.01%	1.02%	0.47%	0.94%	0.13%
Combination (⑦)	0.01%	10.72%	9.35%	2.56%	1.42%
Combination (⑧)	0.03%	52.51%	49.02%	13.37%	51.08%
Total (①-⑧)	100.00%	100.00%	100.00%	100.00%	100.00%

2.5.4. Flip-flop Error Propagation Patterns

Even if a flip-flop error results in R- or M-only propagations, it may not be directly modeled by a single bit-flip error injection into the register-file or program variable. This is because the resulting differences in the register-file or memory (*propagation pattern*) may not be the same as a single-bit error. In this section, we show that only small portions of R- or M-only flip-flop errors are directly modeled by single-bit error injections into the register-file or memory due to the propagation patterns of flip-flop errors.

To have the same resulting differences in the register-file (memory) as single bit-flip error injections using high-level error injection techniques, the error propagation detected by the propagation checker should be a single-bit mismatch in the data input to the register-file (memory), and that should be the only one error propagation during the execution. We refer this propagation pattern as the *single-bit-data only propagation*, or *SBD*. The rest of propagation patterns generally do not result in only a single-bit difference in the register-file or memory (details in Sec. 2.5.4.1).

For the LEON3 processor, the SBD propagation pattern accounts for 39.20% and 7.34% of R-only and M-only propagations, respectively. Considering the percentages of R-only and M-only propagations themselves (Fig. 2.5), R-only SBD accounts for only 2.30% of all flip-flop error injections (or 12.65% of all non-masked flip-flop error injections), and M-only SBD accounts for only 0.11% of all flip-flop error injections (or 0.63% of all non-masked flip-flop error injections).

Similarly, R- or M-only SBD accounts[6] for small portions of flip-flop error injections for the IVM processor either. For the IVM processor, the SBD propagation pattern accounts for 31.82% and 5.79% of R-only and M-only propagations, respectively. R-only SBD accounts for only 0.42% of all flip-flop error injections (or 3.06% of all non-masked flip-flop error injections), and M-only SBD accounts for

[6] The error injection platform for the IVM processor is slightly different from the error injection platform for the LEON3 processor [Cho 13], and the error propagation checkers for the IVM processor do not directly compare the signals (data input, write enable, etc.) to the register-file and memory. Instead, an error propagation is detected if a flip-flop error results in differences in the register-file or memory contents. For example, the SBD propagation pattern is detected if the resulting difference in the register-file or memory is a single-bit mismatch. The I-checker detects incorrect instruction fetches by monitoring the program counter value.

only 0.28% of all flip-flop error injections (or 2.04% of all non-masked flip-flop error injections).

2.5.4.1. Flip-flop Error Propagation Pattern Details

The resulting effects of R-only (M-only) flip-flop errors may not be captured by injecting only a single bit-flip into the register-file (memory) due to the following reasons:

1. R- (or M-) checker can trigger multiple times (**Multi-instance** propagation).
2. Even if the R- (or M-) checker triggers only once (**Single-instance** propagation), the following cases can happen:
   - 2a. Multiple bits of the data input to the register-file (or memory) can mismatch (**Multi-bit** propagation).
   - 2b. Write address input mismatches.
   - 2c. Write enable signal mismatches.
   - 2d. Read address input mismatches.
   - 2e. Read enable signal mismatches.
   - 2f. Combinations of 2 or more of cases 2a-2e. Situations resulting in a single-bit mismatch in the data input to the register-file (or memory) together with any one or more of cases 2b-2e are also included in this category.

The resulting effects of these non-SBD propagations can be captured by single bit-flip error injections only under limited circumstances, e.g., a write enable signal

mismatch pattern when the previous value of the target and the current data input (to the register-file or memory) happens to have only a single-bit difference between them.

For the LEON3 processor, we show the breakdown of the detailed error propagation patterns for the R- and M-only flip-flop errors (Table 2.5). For most of the error injection runs with the R-only propagation type, the R-checker triggers only once (single-instance propagation; 70.67% of R-only flip-flop errors). However, not every single-instance propagation has the SBD propagation pattern. Single-instance propagations that may not be directly modeled by a single-bit error injection into the register-file or program variable (cases 2a-2f) account for 31.47% of R-only propagation types. As aforementioned in Sec. 2.5.4, flip-flop errors with the SBD propagation pattern is only 39.20% of R-only flip-flop errors. Multi-instance propagation (case 1) is fairly common for M-only propagation (53.73%). Also, cases 2a-2f account for 38.93% of M-only propagations. For M-only flip-flop errors, the SBD propagation pattern accounts only for 7.34%.

Table 2.5. Observed rates of flip-flop error propagation patterns for R- and M-only propagation types for the LEON3 processor.

Propagation pattern \ Propagation type		R-only	M-only
1. Multi-instance propagation		29.33%	53.73%
2. Single-instance propagation	2a	7.41%	2.10%
	2b	6.51%	5.47%
	2c	13.40%	19.79%
	2d	1.29%	10.12%
	2e	2.33%	0.00%
	2f	0.53%	1.46%
SBD		39.20%	7.34%

2.5.5. Equivalent Error Candidates

It is possible that a single error in the register-file or program variable during **some** clock cycle *c* can produce the same behavior (especially at the end of program execution) as a flip-flop error which results in error propagation patterns other than R- or M-only SBD.

For example, suppose that a flip-flop error propagates to register *A* at cycle *i* and corrupts the value of register *A* from 0x10 to 0x11. Later, the flip-flop error results in another propagation (multi-instance propagation) to register *B* at cycle *j,* which corrupts the value of register *B* from 0x10 to 0x01 (multi-bit propagation). This propagation pattern does not correspond to R-only SBD. However, for some cases, the resulting behavior from a flip-flop error may be produced by injecting an error into the

register-file or memory. Continuing from our previous example, consider an application that performs an XOR operation using registers *A* and *B* as operands, and stores the result into register *C*. If registers *A* and *B* had the erroneous values due to the flip-flop error (i.e., 0x11 in register *A* and 0x01 in register *B*), the result of the XOR operation that will be stored in register *C* is different from that under error-free execution (error-free value: 0x0 vs. erroneous value: 0x10). If the application uses the value of register *C* to determine its behavior (e.g., ***if*** *C* > 0 ***then*** call task1 ***else*** call task2), the application might produce erroneous operations. Although this flip-flop error does not correspond to R-only SBD propagation, the same erroneous application behavior can be produced by injecting a single bit-flip into register *C*.

Such a "high-level" error may be referred to as an *equivalent error candidate*. This concept is related to fault equivalence in digital testing [McCluskey 71]. The application-level effect of a given flip-flop error can be produced by injecting an equivalent high-level error. For a high-level error injection model to be accurate, there must exist many equivalent "high-level" error candidates for *each* flip-flop error. Otherwise, it will be highly unlikely that an injected high-level error at a randomly-chosen location during a randomly-chosen clock cycle will match the effect of a given flip-flop error.

There could be various definitions of application-level behaviors to determine whether two application executions are considered to have the "same" behavior or not. For example, having the same type of application outcome (e.g., outcome type classification in Sec. 2.3) can be regarded as having the same behavior. In this case, the probability of having equivalent error candidates can be derived using the error

injection results presented in Sec. 2.3 and 2.4[7]. In this section, we consider two executions to have the same application-level behavior if they have the same register-file and memory states at the end of the application execution[8]. Using this definition, for instance, two erroneous application executions with the OMM type outcome are considered different behaviors if their (corrupted) output files do not exactly match each other. Our analysis in this section shows that there is a very low chance that a randomly-chosen single-bit error injection into the register-file or memory can be an equivalent error candidate for a flip-flop error.

We performed a set of error injection runs to save and compare the register-file and memory states at the end of application execution (*last architectural state*) for the LEON3 processor. For each error injection technique (Flip-flop, RegU, RegW, VarU, and VarW), we collected the last architectural states from more than 40,000 error injection runs using the *crafty* application. For each injected flip-flop error, we compared its last architectural state to each of the last architectural states obtained from high-level error injections to estimate the probability of obtaining equivalent error candidates using high-level error injection technique (*matching probability*). For this comparison, we exclude the error injection runs that result in the Vanished outcome type because any application run with the Vanished outcome type will always have the same architectural state at the end of the execution (by definition).

[7] For example, in the LEON3 results, the OMM outcome rate for the crafty application is 1.3% for flip-flop error injection and 6.7% for RegW error injection. In this case, those flip-flop soft errors (that result in the OMM outcome type) can be considered to have the same behavior as an error injection run (using RegW) if the application outcome from the RegW error injection run happens to be OMM as well (6.7% of RegW error injection runs). Therefore, each flip-flop soft error that results in the OMM outcome type has a 6.7% chance of obtaining equivalent error candidates by using RegW.

[8] For the UT outcome type, we consider the cycle the application stops execution (e.g., processor halt or reset) as the end of application execution. For the Hang type outcome, we consider the cycle at which the application reaches the timeout limit as the end of the application.

Figure 2.7 shows the observed probabilities (horizontal axis) of obtaining equivalent error candidates using high-level error injection for a flip-flop error. The vertical axis shows how many flip-flop errors have such probabilities for obtaining equivalent error candidates[9]. The leftmost bar (black bar) shows that for 97.3% of flip-flop error injection runs (one flip-flop error injected per run), we were not able to find any match with the error injection runs performed using one of the high-level error injection techniques (RegU, RegW, VarU, or VarW), i.e., no equivalent error candidates were found. The low probabilities further confirm that randomly injecting single bit-flips into the register-file or memory has very little chance of resulting in the same behavior as flip-flop error injections.

[9] Each bar represents the percentage of flip-flop errors that have probabilities belonging to the interval of the bar. The interval of a bar that is drawn in between *x* and *y* of the horizontal axis is (x,y] (one exception is the leftmost bar, which represents the percentage of flip-flop errors that have zero probability of obtaining equivalent error candidates). For example, the second bar from the left (drawn in between 0.00% and 0.01%) shows that 0.8% of flip-flops have probabilities that are greater than 0.00% *and* less than or equal to 0.01%.

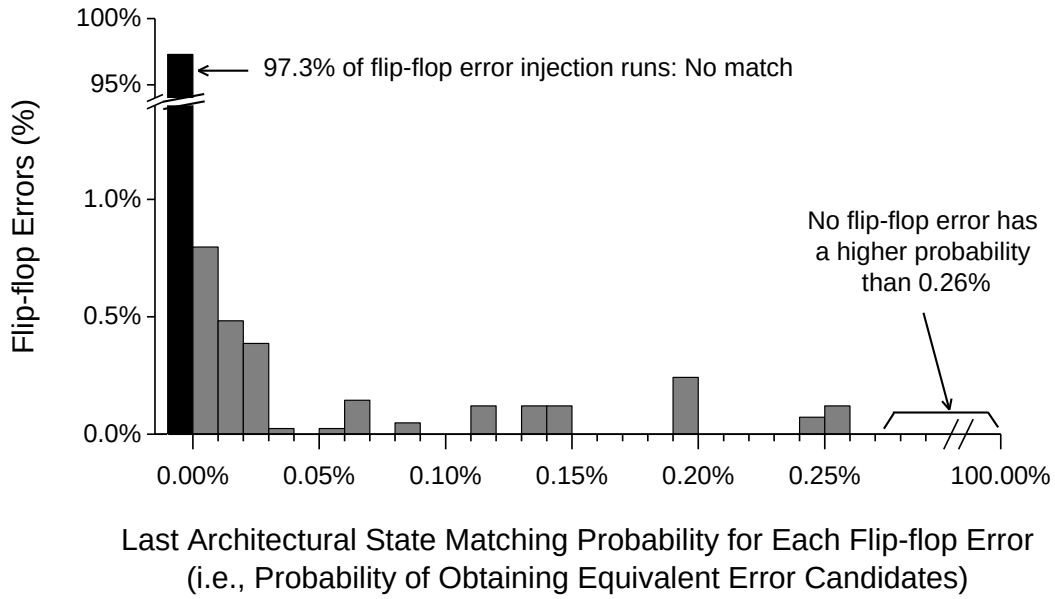


Figure 2.7. Observed probabilities of obtaining equivalent error candidates using high-level error injection for a given flip-flop error.

2.5.6. Error Propagations to "Other" Software-visible States

In Sec. 2.5.4, we showed that the majority of flip-flop errors do not correspond to R- or M-only SBD. Also, in Sec. 2.5.5, we showed that there is a very small chance that a random single bit-flip into the register-file or memory results in application-level behaviors that are equivalent to that of a flip-flop error. Imitating the behavior of flip-flop errors by injecting (multiple) errors into the register-file or memory that corresponds to all error propagations resulting from the flip-flop error may involve rigorous error modeling and complicated error injection methodologies [Mirkhani 14]. In this section, we explore an alternative way to improve the accuracy of high-level error injection techniques.

There exist other software-visible architectural states in addition to the register-file and memory that can also affect application execution, e.g., program counter and integer condition codes[10]. We refer these states as *"Other"* states. The Other states are usually not targeted by high-level error injection techniques discussed in Sec. 2.2.3. If the Other states have erroneous values that are different from error-free execution (*Other state corruptions*), the corrupted other states can result in subsequent error propagations to the register-file or memory. For example, due to a corrupted integer condition code, a conditional branch (e.g., branch-on-equal instruction in the SPARC instruction set architecture of the LEON3 processor) can take a different execution path; an application executed along an incorrect execution path may result in multiple mismatching register or memory values (when compared to those along the correct execution path). In such a case, an error injection technique that directly injects errors into the corresponding Other states (the integer condition code in this example) can result in the same behavior instead of injecting multiple register-file or memory errors.

In the LEON3 processor, the Other states are stored in flip-flops (*Other state flip-flops*). The Other state flip-flops can get corrupted in the following ways:

1. A soft error directly causes a bit-flip in one of the Other state flip-flops.
2. A soft error in a flip-flop (not belong to the Other states) affects one or multiple Other state flip-flops.

[10] Integer condition codes indicate the resulting conditions of the recent operation, e.g., overflow or carry bits.

3. A flip-flop soft error, propagated to the register-file or memory, results in subsequent propagations to the Other states. Already corrupted Other states may also result in (subsequent) corruptions of Other states.

We implemented another error propagation checker (in addition to the propagation checkers in Fig. 2.4) to detect Other state corruptions (Fig. 2.8). Similar to the error propagation tracking mechanism for the register-file and memory, we do not consider subsequent propagations (case 3). The *F-checker (Flip-flop-checker)* detects errors propagated to the Other states by comparing inputs to Other state flip-flops from the erroneous core vs. the golden core. The F-checker also detects flip-flop soft errors directly injected into the Other state flip-flops. In this setup, unlike the error propagation tracking mechanism used in the prior sections (Fig. 2.4), the golden core and the erroneous core share the same Other state flip-flops. These flip-flops are always updated using the inputs from the erroneous core; the inputs from the golden core are used by the F-checker for error propagation detection. By sharing the Other state flip-flop contents between the two cores, the F-checker detects only initial propagations to the Other states, not subsequent propagations (similar to the R-, M-, and I-checkers in Fig. 2.4).

The list of architectural states that belong to the Other states for the LEON3 processor (detected by the F-checker) is shown in Table 2.6. For flip-flop errors that result in at least one F-checker firing during the execution, Table 2.6 also shows how often such a flip-flop error results in corruptions in each of the architectural states (*corruption rate*).

This setup also includes a *backup golden core* and the *backup propagation checkers*. The backup checkers (backup R-, M-, and I-checkers) are used for identifying subsequent propagations to the register-file and memory that are caused by corrupted Other states. The backup golden core has its own Other state flip-flops (Table 2.7). If corrupted Other states result in subsequent error propagations to the register-file or memory, there would be mismatches between the erroneous core and the backup golden core, and the backup propagation checkers detect those mismatches.

Because the backup golden core does not share the (corrupted) Other states with the erroneous core (while sharing the register-file and memory), the mismatches detected by the backup propagation checkers are either subsequent propagations from the Other states or initial propagations from the erroneous core. In contrast, the golden core and the erroneous core will not show mismatches for subsequent error propagations resulting from Other state corruptions. By comparing the differences between the R-, M-, and I-checker firings and their corresponding backup checker firings at the same cycle, we can identify the subsequent propagations resulting from Other state corruption (Table 2.8 summarizes various checker firing scenarios).

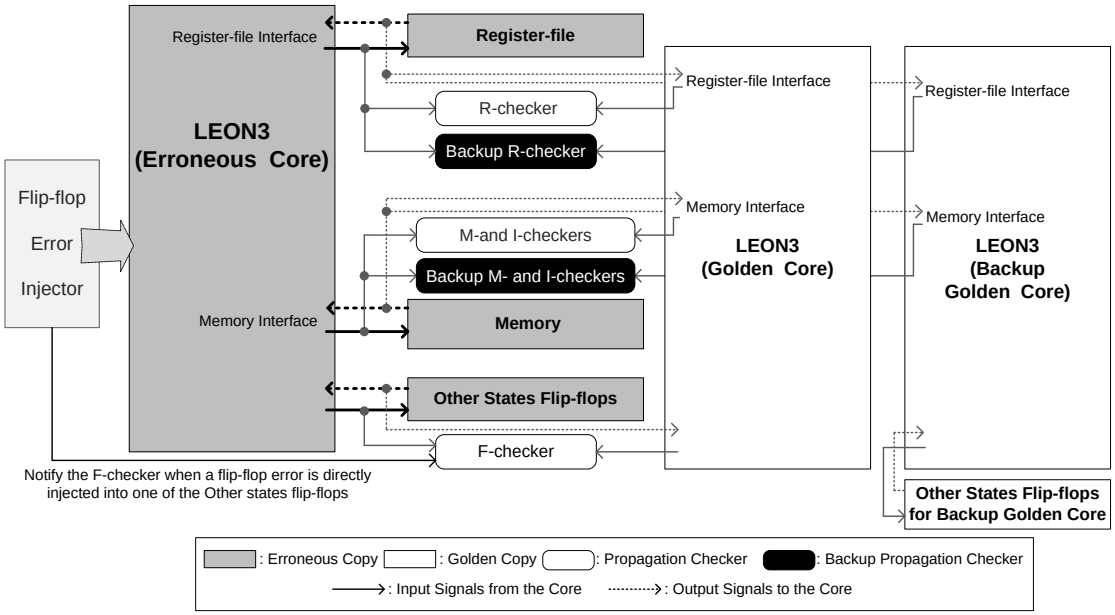


Figure 2.8. Error propagation detection logic for the Other states.

Table 2.6. List of architectural states belong to the Other states and their corruption rates[11] for the LEON3 processor.

State	Corruption rates
Program counter corruption	40.02%
Integer condition codes	15.59%
Peripheral states	3.43%
Supervisor mode	3.59%
Processor internal interrupt mode	17.85%
Register window	26.43%
Multiplier result (Y-reg.)	20.04%

[11] The corruption rates do not add up to exactly 100% because for some flip-flop errors, the F-checker can trigger multiple times for multiple architectural states belong to the Other states.

Table 2.7. States shared by processor cores in Fig. 2.8. *Private*: The core has its own state (not shared with other cores). *Shared*: The same (erroneous) state is used by multiple cores. The shared states are updated with the inputs from the erroneous core only.

		Erroneous core	Golden core	Backup golden core
Non-software-visible states		Private	Private	Private
Software-visible architectural states	Register-file	Shared	Shared	Shared
	Memory (includes instructions)	Shared	Shared	Shared
	Other states	Shared	Shared	Private

Table 2.8. Various backup checker firing scenarios.

		Original R-, M-, or I-checker	
		Triggers	Does not trigger
Corresponding backup checker (backup R-, M-, or I-checker)	Triggers	Initial propagation from flip-flop error.	Subsequent propagation from Other state corruptions.
	Does not trigger	Initial propagation from flip-flop error[12].	No propagation detected.

Using the error propagation logic in Fig. 2.8, we ran 440,000 error injection runs (40,000 error injection runs per application). Figure 2.9 shows a Venn diagram of various propagation types including Other state corruptions detected by the F-checker. Due to the added F-checker, the propagation types are different from the propagation types discussed in Sec. 2.5.2.

1. **Masked:** The flip-flop error triggers none of the R-, M-, I-, and F-checkers.

[12] In general, a backup checker (backup R-, M-, or I-checker) triggers when the corresponding original checker (R-, M-, or I-checker) triggers. However, for some rare cases, an original checker may detect a mismatch when the corresponding backup checker does not trigger. Consider the following flip-flop error behavior:

1. A flip-flop error corrupts the Other states (these corrupted Other states are shared between the erroneous core and the golden core).
2. Even after the Other states corruption, the flip-flop error persists in the non-software-visible states of the erroneous core (i.e., corrupted non-software-visible states).

In such a situation, all three cores have different states (i.e., erroneous core: corrupted Other states and corrupted non-software-visible states, golden core: corrupted Other states only, and backup golden core: no corrupted states). If an original checker triggers in this situation, the mismatch is caused by the corrupted non-software-visible states in the erroneous core, which are not shared with the golden core. Since those corrupted non-software-visible states are not shared with the backup golden core either, the corresponding backup checker may also trigger. However, the difference between the erroneous core and the backup golden core also includes the corrupted Other states of the erroneous core. If the combination of the corrupted Other states and the corrupted non-software-visible states causes the erroneous core to behave in the same way as the backup golden core, the corresponding backup checker does not trigger (when the original checker triggers). Since the corrupted Other states alone do not cause such behavior of the erroneous core, this behavior of the erroneous core is also classified as an initial propagation (i.e., resulting from the flip-flop errors in non-software-visible states).

2. **No-Other-corruption**: The flip-flop error does not result in any Other state corruption, i.e., only one or multiple of the R-, M-, and I-checkers trigger.
3. **Other-only (O-only)**: The flip-flop error only results in Other state corruptions, i.e., only the F-checker triggers (once or multiple times) during program execution. As a result of the corrupted Other states, there could be subsequent propagations to the register-file and memory.
4. **Combination**: The flip-flop error results in Other state corruptions *and* error propagations to the register-file or memory (detected by the F-checker and one or multiple of the R-, M-, and I-checkers). Errors detected by the R-, M-, or I-checkers do not result from the propagation of errors detected by the F-checker. It may not be possible to model error propagations to the register-file and memory resulting from these flip-flop errors by error injections into the Other states only.

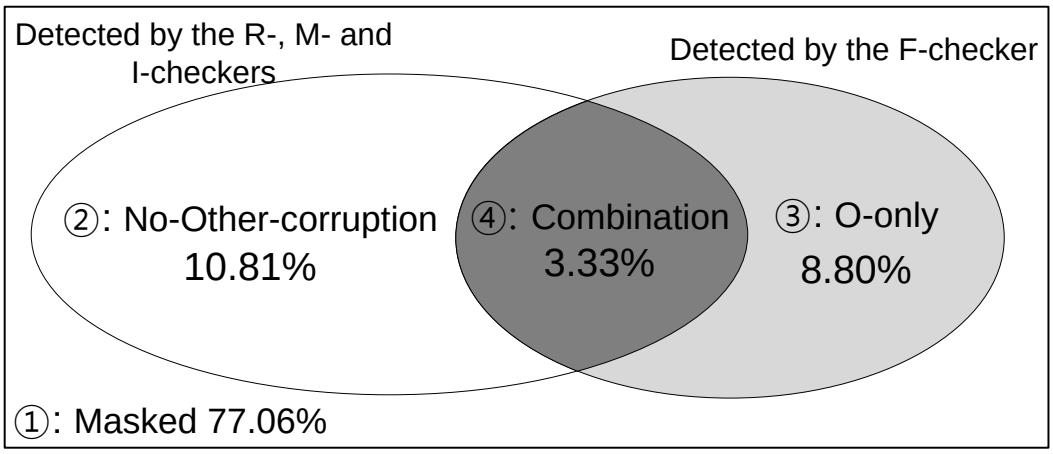


Figure 2.9. Observed rates of flip-flop error propagation types including Other state corruptions.

Flip-flop errors with the O-only propagation type (③, 8.80% of injected flip-flop errors) can be modeled by directly injecting errors into the corresponding Other states using architectural- or software-level error injection techniques. Moreover, if the Other state corruptions caused by an O-only flip-flop error is only a single-bit corruption in the Other states, the resulting effects of the flip-flop error can be modeled using a (simple) single bit-flip error injections into the Other states[13].

Table 2.9 shows the observed rates of the following *subsequent propagation patterns* to the register-file and memory (detected by the backup R-, M-, and I-checkers) resulting from O-only flip-flop errors (③ in Fig 2.9).

[13] Combinations of error injections into the register-file or memory *and* error injections into the Other states may be used for better accuracies. In this section, for simplicity, we discuss the potential benefits of error injections into the Other states only.

P1. **No propagation:** The Other state corruptions caused by the flip-flop error do not result in any subsequent propagation to the register-file or memory, i.e., result in the Vanished application outcome type.

P2. **Single-bit-data only (SBD):** The subsequent propagations resulting from the Other state corruptions correspond to R- or M-only SBD. These flip-flop errors are modeled by single bit-flip error injections into the register-file or memory.

P3. **Non-SBD:** The Other state corruptions result in register-file or memory propagations that have propagation patterns 1 or 2a-2f in Sec. 2.5.4. These flip-flop error propagations are not identically modeled by single bit-flip error injections into the register-file or memory.

The results show that Non-SBD pattern (*P3*) is more frequently observed vs. the SBD pattern (*P2*) (4.49% vs. 0.69% of injected flip-flop errors or 51.02% vs. 7.86% of O-only flip-flop errors). This means that for the majority of O-only flip-flop errors (that result in Non-SBD subsequent propagations), single bit-flip error injections into the register-file or memory do not identically model the resulting error propagations to the register-file or memory (that are caused by corrupted Other states).

To quantify the portions of O-only flip-flop errors that can be modeled by injecting a *single bit-flip* into the Other states only, Table 2.9 also shows the following breakdown of the observed patterns of resulting mismatches (corruptions) in the Other states detected by the F-checker (*Other state corruption patterns*).

C1. **Single-bit:** The F-checker detects only a single-bit mismatch[14] in the Other state flip-flops only once during the application execution.

C2. **Multi-error:** The F-checker detects multi-bit mismatches in the Other state flip-flops during a single cycle or multiple cycles (including single-bit mismatches over multiple cycles).

Most of the O-only flip-flop errors result in the Single-bit corruption (*C1*) in the Other states (94.24% of O-only flip-flop errors or 8.29% of all injected flop-flop errors). The resulting effects of these flip-flop errors can be modeled by injecting a single error directly into the software-visible Other states (using a high-level error injection technique).

[14] Depending on the processor implementation, the Other states visible from the software and the actual flip-flop contents may or may not have one-to-one correspondence on each bit (bit-to-bit correspondence). The LEON3 processor implementation does have bit-to-bit correspondences for all Other states (with the exception of hard-wired bits to '0' or '1', which are not affected by flip-flop soft errors). If there is no bit-to-bit correspondence on some states, even a single-bit mismatch in the flip-flop may be treated as the multi-error corruption pattern.

Table 2.9. Observed rates of Other state corruption patterns and subsequent propagation patterns to the register-file and memory.

Other state corruption pattern	Subsequent propagation patterns (to the register-file and memory) resulting from Other state corruptions	Observed rates (% of all injected flip-flop errors)	Observed rates (% of all O-only flip-flop errors)
C1. Single-bit	*P1*. No propagation	3.47%	39.46%
	P2. SBD	0.69%	7.86%
	P3. Non-SBD	4.13%	46.92%
	Total (*C1*. *P1-P3*)	8.29%	94.24%
C2. Multi-error	*P1*. No propagation	0.15%	1.66%
	P2. SBD	0.00%	0.00%
	P3. Non-SBD	0.36%	4.10%
	Total (*C2*. *P1-P3*)	0.51%	5.76%
Total		8.80%	100%

2.5.7. Results Cross-Check

In this section, we analyze the outcomes obtained from flip-flop error injections resulting in R- and M-only SBD (Sec. 2.5.2 and 2.5.4) versus outcomes obtained from RegW and VarW error injections. We expect the inaccuracy of outcome rates resulting from RegW (VarW) error injections to be lower when compared to flip-flop error injections resulting in R-only (M-only) SBD, rather than the entire set of flip-flop error injections (Sec. 2.3 and 2.4). While this analysis can be somewhat conservative, given the discussion on equivalent error candidates in Sec. 2.5.5, it

provides insights into inaccuracies of high-level error injections resulting from the fact that they model a small fraction of all flip-flop errors. As discussed in Sec. 2.3, RegU and VarU error injections may inject errors into locations not accessed during the rest of the execution. Hence, we focus on RegW and VarW error injections.

Tables 2.10-2.13 report the estimated inaccuracy of RegW and VarW error injections for a given outcome *x* using the following expression.

$$\frac{|G.Mean(x,H) - G.Mean(x,F)|}{G.Mean(x,F)} \quad (2.2)$$

where *H* is the RegW or VarW error injection technique, *F* corresponds to flip-flop error injections being compared to (either the entire set of flip-flop error injections or flip-flop error injections resulting in R- or M-only SBD), and *G.Mean*(*x, t*) is the geometric mean of normalized outcome rates obtained using (2.1). The accuracy improvements with SBD are also estimated in Tables 2.10-2.13.

As expected, accuracies improve when high-level error injection results are compared with respect to flip-flop errors resulting in SBD: by 2.17× (4.29× for IVM) on average (geometric mean) for RegW and 4.83× (2.89× for IVM) on average for VarW. For the OMM outcome type, the accuracies are improved for more than an order of magnitude for both processors.

Table 2.10. Inaccuracies of outcome rates resulting from the RegW error injection techniques for the LEON3 processor.

Outcome	RegW inaccuracies		
	vs. flip-flop error injections (i)	vs. flip-flop error injections resulting in R-only SBD (ii)	Accuracy improvement (i / ii)
Vanished	11.6%	11.9%	0.97×
ONA	126%	29.1%	4.3×
OMM	416%	35.2%	11.8×
UT	14.9%	32.9%	0.45×
Hang	122%	56.5%	2.16×
		Geometric mean	2.17×

Table 2.11. Inaccuracies of outcome rates resulting from the VarW error injection techniques for the LEON3 processor.

Outcome	VarW inaccuracies		
	vs. flip-flop error injections (iii)	vs. flip-flop error injections resulting in M-only SBD (iv)	Accuracy improvement (iii / iv)
Vanished	40.1%	39.9%	1.0×
ONA	468%	35.6%	13.2×
OMM	1065%	49.3%	21.6×
UT	4.18%	34.9%	0.12×
Hang	199%	2.6%	77.0×
		Geometric mean	4.83×

Table 2.12. Inaccuracies of outcome rates resulting from the RegW error injection techniques for the IVM processor.

Outcome	RegW inaccuracies		
	vs. flip-flop error injections (i)	vs. flip-flop error injections resulting in R-only SBD (ii)	Accuracy improvement (i / ii)
Vanished	8.86%	3.72%	2.38×
ONA	20.4%	43.7%	0.47×
OMM	121%	5.14%	23.5×
UT	69.6%	9.73%	7.15×
Hang	108%	13.9%	7.77×
		Geometric mean	4.29×

Table 2.13. Inaccuracies of outcome rates resulting from the VarW error injection techniques for the IVM processor.

Outcome	VarW inaccuracies		
	vs. flip-flop error injections (iii)	vs. flip-flop error injections resulting in M-only SBD (iv)	Accuracy improvement (iii / iv)
Vanished	11.4%	3.93%	2.90×
ONA	53.5%	53.4%	1.00×
OMM	340%	11.8%	28.8×
UT	16.4%	10.8%	1.52×
Hang	88.6%	55.9%	1.58×
		Geometric mean	2.89×

2.6. Conclusion

Existing high-level error injection techniques, that inject single-bit errors at randomly-chosen register and memory locations during randomly-chosen clock cycles, can be highly inaccurate when compared to flip-flop error injection techniques. This chapter demonstrates this point for the LEON3 in-order processor core as well as for a complex out-of-order Alpha-like IVM processor core. This chapter also quantifies the causes of these inaccuracies through a detailed analysis of error propagation through various layers of the system stack. The presented results provide insights that can potentially help us create new classes of high-level error models with significantly higher accuracies. While the feasibility of high-level error models that are accurate for any arbitrary digital system is unclear, one can possibly use our results to derive accurate high-level error models that are tailored for certain families of digital systems.

This chapter focuses on the accuracy aspects of existing high-level error injection techniques. However, depending on the application, accuracy is not necessarily a requirement. For example, an inaccurate error injection technique can be very useful as long as it is effective in driving the correct design decisions for building robust systems. One such example exists in digital system testing literature. Stuck-at faults are highly inaccurate in modeling actual manufacturing defects, but are highly effective as test metrics that drive automatic test pattern generation and design for testability techniques [McCluskey 00]. Future research must explore and quantify this aspect of high-level error models.

Chapter 3

Mixed-mode Simulation Platform for Soft Error Simulation

3.1. Introduction

Radiation-induced soft errors pose a major challenge to building robust systems using complex System-on-Chips (SoCs). Although the soft error rate at the device level (e.g., SRAM cell or latch) stays roughly constant or even decreases over technology generations, the system-level soft error rate increases as more devices are integrated into SoCs [Mitra 14, Seifert 10, 12].

Uncore components[1], such as cache controllers, DRAM controllers and I/O controllers, are increasingly important because their overall area footprint and power consumption in SoCs are comparable to that of processor cores [Gupta 12, Li 13]. The

[1] Also be referred to as "nest," "outside-core," or "northbridge". In this dissertation, we use this term to refer to components that are not processors or accelerators.

need for studying soft errors in uncore components has been pointed out in the literature [Mukherjee 05, Quinn 13]. While there are many studies on soft errors in processor cores (e.g., [Cho 13, Ramachandran 08, Wang 04]), few have studied soft errors in uncore components. The lack of such studies can be attributed to the difficulties in modeling large-scale SoCs (with multiple processor cores and multiple uncore components) for the following reasons.

1. Uncore studies should model the entire SoC because uncore components interact with processor cores and other uncore components. Modeling only a part of the system may not capture uncore behaviors accurately.
2. Studying system-level effects of soft errors requires real-world applications. This becomes more relevant in the context of cross-layer resilience, where multiple error resilience techniques from various layers of the system stack are combined to achieve cost-effective solutions [DeHon 10, Mitra 10, 14].
3. For statistically significant results, a large number of error injection samples are required. For example, when observing a certain outcome rate, more than 40,000 samples are required to achieve ±0.1% accuracy with 95% confidence when the observed rate is 1%[2].

Such requirements demand high-throughput error simulation or emulation platforms. RTL simulators that model detailed error behaviors are extremely slow. For example, RTL simulation of an out-of-order, superscalar processor core achieves less

[2]This assumes the normal approximation of the binomial distribution, similar to the confidence interval used in [Choi 90].

than a thousand cycles per second [Maniatakos 11b]. High-level simulators, on the other hand, achieve much faster simulation times [Simics]. However, naïvely injecting errors into abstracted high-level layers without adequate low-level details can result in highly inaccurate results (e.g., results in [Cho 13] for processor cores).

Existing uncore error studies are limited to very small designs (e.g., private L1 cache and bus controller in a design with a single processor core [Bailan 10]) or rely on fast high-level simulators without low-level details (e.g., error injections into primary input and output signals in [Graham 09, Lin 06]). While radiation testing can be used to study overall soft error resilience of a design [Bender 08, Sanda 08], it is only available after the chip is produced. Also, quantifying vulnerabilities of various on-chip components can be difficult using radiation testing due to limited observability.

In this chapter, we make the following contributions:

1. We present a simulation platform that is capable of simulating large-scale SoCs while modeling detailed flip-flop soft errors. Compared to RTL-only simulation, this platform achieves over 20,000× speedup.

2. We present the first study of system-level effects of soft errors in uncore components in a large-scale OpenSPARC T2 SoC with 500 million transistors, eight processor cores, and many uncore components [OpenSPARC]. We report quantified results on the effects of soft errors in L2 cache controllers, DRAM controllers, crossbar interconnects, and PCI Express I/O controllers. We show that

soft errors in uncore components can have significant reliability impact comparable to that of processor cores.

3.2. Mixed-mode Soft Error Simulation Platform

To analyze the effects of uncore soft errors in large-scale SoCs, we created a mixed-mode platform that combines two simulation platforms (sometimes referred to as *co-simulation* in design validation literature [Benini 03]). The target uncore component is simulated using an RTL simulator to model soft error behaviors with low-level details, while the rest of the system is simulated using a high-level simulator. Our mixed-mode platform is different from existing co-simulation-based studies on error behaviors for the following reasons:

1. [Li 09, Ejlali 03] use co-simulation to study errors in small combinational logic blocks, such as the ALU or the decoder module with only a few hundred gates, inside a processor core. To correctly model how soft errors in flip-flops behave inside an uncore component, we model an entire uncore component (more than 100K gates) using RTL, and ensure that state transfer between the RTL simulator and the high-level simulator does not become a performance bottleneck.
2. [Goswami 97, Kalbarczyk 99] profile high-level effects resulting from low-level errors, and use the statistical information for quick error simulations. Profiled error behaviors may not reflect subsequent error propagations due to interactions with the rest of the system (e.g., a flip-flop error in a module may result in multiple erroneous interactions with other components [Cho 13]). We model

how the error interacts in a chip by simulating its behavior at the entire chip level until all the effects from the injected error have been fully modeled.

3. [Wang 04] uses two simulators at two different levels of abstraction to simulate a processor core, but only one of the simulators is used at a given point in time. This approach requires transferring the entire system state between the simulators. In our platform, we utilize low-level simulation only for the target uncore component. Our approach reduces state transfer and low-level simulation overheads.

FPGA emulation platforms can achieve faster speeds compared to RTL simulations while modeling low-level details [Asaad 12, Schelle 10]. However, to model an entire SoC, the design may need to be mapped on multiple FPGA chips. This is because the area required for the FPGA implementation of a design can be an order of magnitude greater than an ASIC implementation (for the same technology generation) [Kuon 07]. As a result, limited inter-FPGA I/O bandwidth can limit the overall emulation speed to only a few MHz [Hauck 07].

3.2.1. Mixed-mode Platform Simulation Modes

Our platform operates in two modes:

1. **Accelerated mode** (Fig. 3.1a**):** All components on the chip, including processor cores and uncore components, are simulated using the Simics instruction-set simulator [Simics]. The uncore components are simulated using high-level

models. Under error-free conditions, they produce the same output signals to processor cores as the actual uncore components (Fig. 3.1a ①). Table 3.1 lists the uncore states modeled by the high-level uncore models (*high-level uncore state*). Flip-flops inside uncore components are not fully modeled in this mode.

2. **Co-simulation mode** (Fig. 3.1b**):** The target uncore component is simulated using an RTL simulator. Processor cores access uncore components by exchanging requests and return packets through the on-chip interconnect (e.g., PCX and CPX packets in OpenSPARC T2). During co-simulation mode, these access packets to and from the uncore component are transferred between the high-level simulator and the RTL simulator (Fig. 3.1b ②). To ensure cycle-level accuracy, the two simulators are synchronized every cycle to ensure transfer of packets between simulators at the correct cycle.

Although the accelerated mode cannot simulate how a soft error behaves at the flip-flop level, high-level models can correctly simulate subsequent behaviors after a flip-flop soft error fully propagates to the high-level uncore state (i.e., no flip-flop or SRAM array inside the uncore component, not included in the high-level uncore state, contains an error).

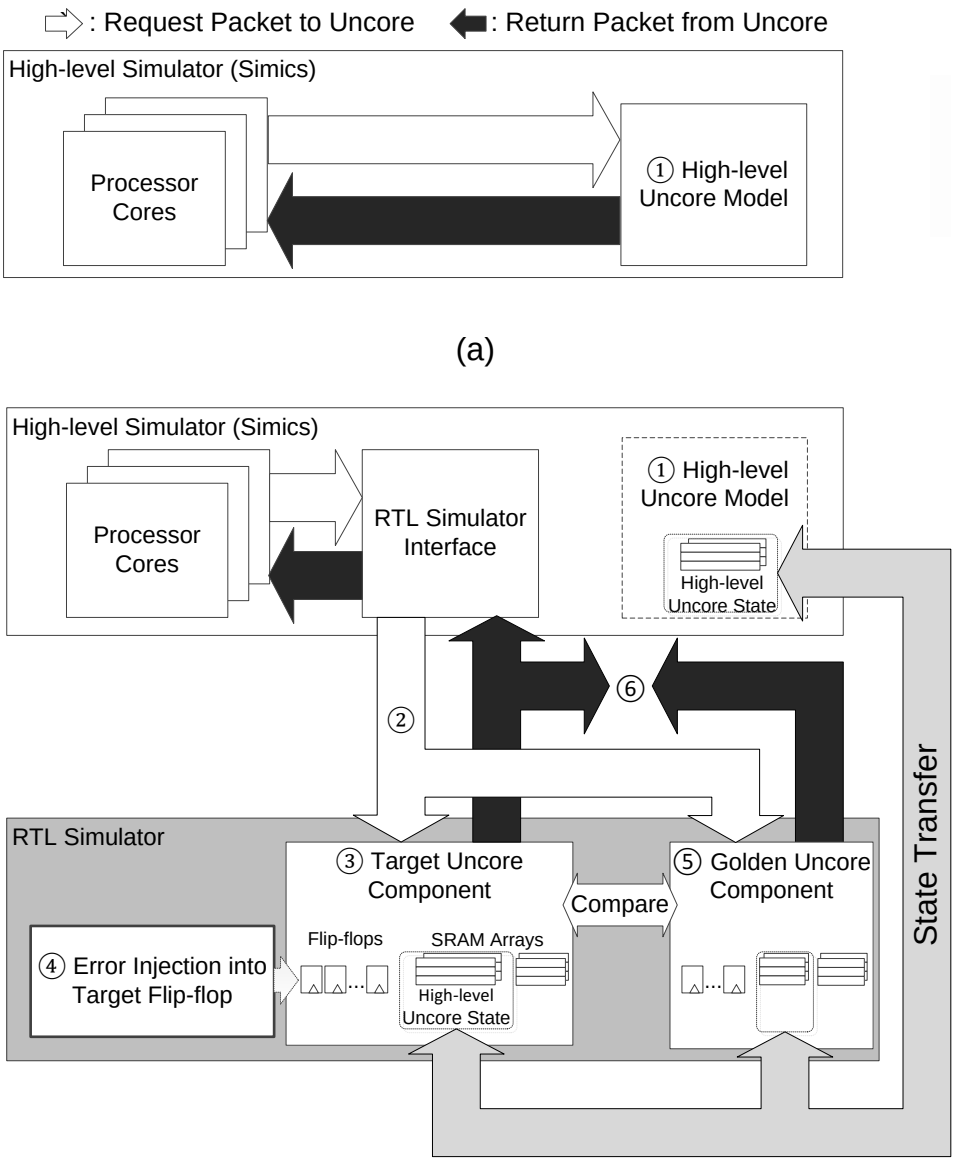


Figure 3.1. Mixed-mode platforms. (a) Accelerated mode. (b) Co-simulation mode.

Table 3.1. High-level uncore states modeled by the high-level uncore models.

Uncore component	High-level uncore states (size per instance)
L2 cache controller	Tag address array (28KB), Cache line state bit array (5KB), Cache data array (512KB), L1 cache directory (2KB)
DRAM controller	DRAM contents (4GB)
Crossbar interconnect	None[3]
PCI Express I/O controller	Transfer buffers (RX: 8KB, TX: 4KB)

3.2.2. Soft Error Injection Methodology

Figure 3.2 shows the flowchart of our uncore error injection methodology using our *mixed-mode platform.* The co-simulation mode is invoked only when soft error injection begins and terminated when the injected error disappears without any remaining error or when the remaining errors can be simulated using the accelerated mode.

[3] The crossbar interconnect only delivers packets between processor cores and L2 cache controllers. Therefore, its states can be reconstructed in the co-simulation mode without modeling a separate high-level uncore state for the crossbar in the accelerated mode.

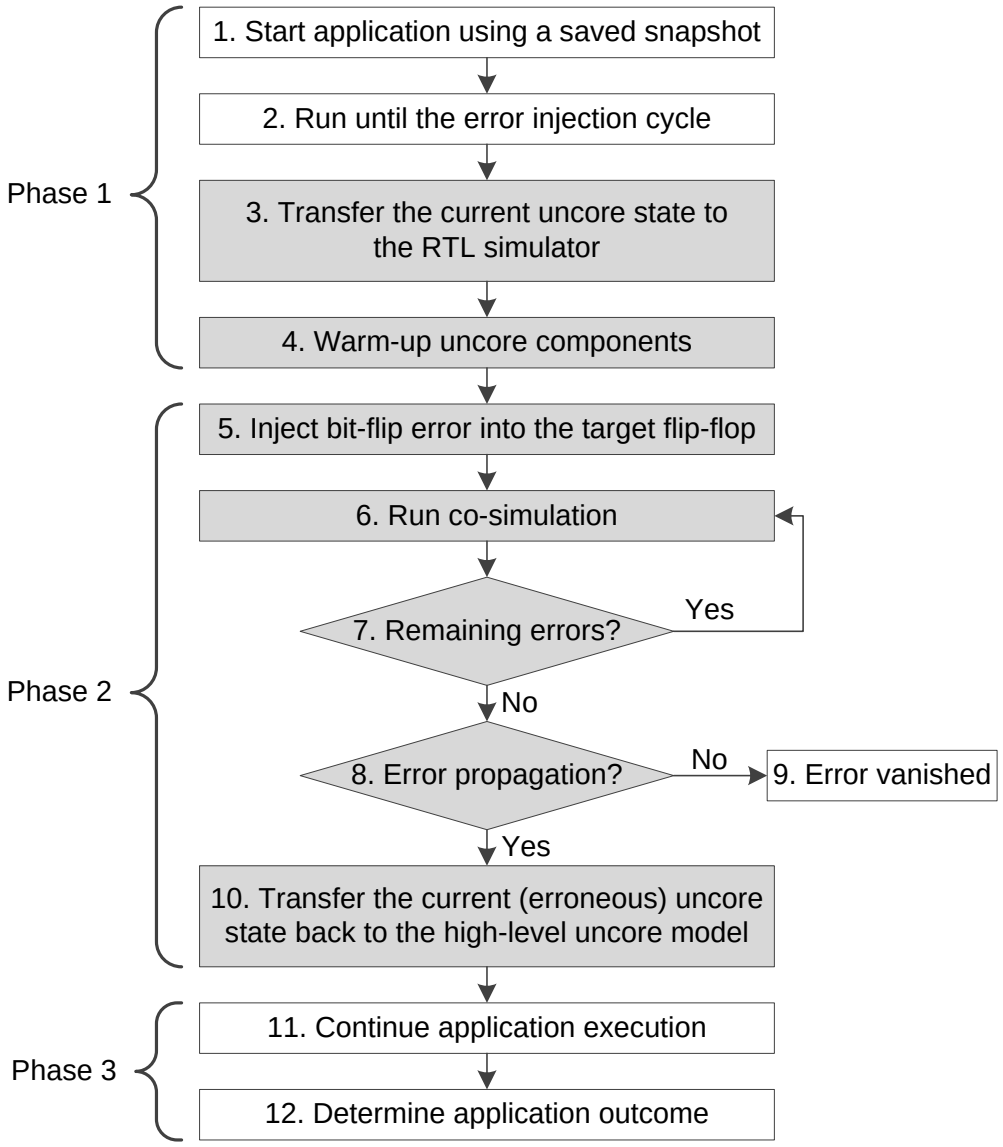


Figure 3.2. Error injection using our new mixed-mode simulation platform. Steps in gray color use co-simulation mode.

Phase 1. Prepare for Error Injection: For each error injection run, an error injection cycle from high-level simulation (in accelerated mode) and a target flip-flop inside the target uncore component are randomly selected. The mixed-mode platform starts

application execution in accelerated mode and simulates the application until the error injection cycle (Fig. 3.2, steps 1 and 2). This step is shortened by starting the simulation using one of the system state snapshots obtained from a one-time, error-free execution of the application in accelerated mode. If the error injection cycle is C_i and the snapshots are created every C_f cycles, the simulation is started using a snapshot created at cycle C_s, where $C_s=\lfloor C_i/C_f \rfloor \times C_f$. For our error injection runs, we created a snapshot every 2 million cycles.

When RTL simulation starts (Fig. 3.2, step 3), high-level uncore states that have been simulated by the high-level model (Fig. 3.1a ①) are transferred to the target uncore component in the RTL simulator (Fig. 3.1b ③). A warm-up period is required before the error injection to correctly restore all microarchitectural states (e.g., flip-flops and small SRAM buffers) that have not been simulated by the high-level model (Fig. 3.2, step 4). The actual warm-up period is randomly selected for each run to avoid injecting errors always after the same number of co-simulation cycles. In our platform, the warm-up period is at least 1,000 cycles, which is enough to reconstruct microarchitectural states for the tested OpenSPARC T2 uncore components (detailed discussion in Sec. 3.4.1).

Phase 2. Inject Error: A bit-flip error is injected into the selected flip-flop (Fig. 3.1b ④, Fig. 3.2, step 5). The platform periodically checks if the accelerated mode can take over the simulation by checking remaining errors in RTL (Fig. 3.2, steps 6-7). This check is done by comparing the values of the storage elements (flip-flops, SRAM arrays) in the target uncore component, where the error is injected (Fig. 3.1b ③), with

the corresponding values in the golden component (Fig. 3.1b ⑤). The golden component is an identical copy of the target uncore component that receives the same input, but simulated without error injection. It is only used for simulation purposes to check when to end the co-simulation mode. The co-simulation mode is no longer needed if the comparison finds no mismatch or all mismatches satisfy one of the following conditions:

1. The mismatch can be directly mapped to high-level uncore states. The subsequent effects can be simulated by using the accelerated mode.
2. The mismatch does not cause any functional difference (e.g., corrupted data field when the associated valid flag is not set; the value will not be used by the application in that case).

Phase 3. Determine Application Outcome: The current uncore state in RTL is transferred back to the high-level model, and the mixed-mode platform continues to run the application to completion in the accelerated mode to determine if the application run results in any erroneous outcome (Fig. 3.2, steps 10-12).

During phase 2, the platform monitors if an injected error has produced erroneous return packets to the processor cores by comparing return packets from the target uncore component to those of the golden uncore component (Fig. 3.1b ⑥). If no erroneous return packet has been detected and the transferred state from the target uncore component matches that from the golden uncore component, the error injection run will result in the same outcome as that of the error-free run. For those cases, the

simulation can stop early without executing the rest of the application in phase 3 (Fig. 3.2, steps 8-9).

3.2.3 Mixed-mode Simulation Performance

The effective simulation throughput of the mixed-mode platform is over 2M cycles/sec, comparable to that of multi-FPGA platforms for large-scale SoCs [Asaad 12, Schelle 10]. Compared to the RTL-only simulation of the OpenSPARC T2 design (up to 100 cycles/sec only [Weaver 08]), we achieve more than 20,000× speedup. By utilizing saved snapshots, steps 1-2 take only 1M cycles on average. Steps 11-12 are executed only for less than 1% of total error injection runs[4]. Table 3.2 summarizes the performance of our mixed-mode platform when simulating an application with cycle length *L* for the OpenSPARC T2 design. For applications with cycle lengths longer than 280M, the throughput is over 2M cycles/sec. Applications with shorter lengths achieve throughput values less than 2M cycles/sec (e.g., the *Radix* application with *L*=120M in Sec. 3.3.2 achieves 1M cycles/sec); however, those applications require shorter simulation times.

$$\text{Throughput} = \frac{Applicaton\ length}{avg.\ simulation\ time} = \frac{L}{70+\frac{L}{4M}} > \text{2M cycles/sec}, \quad (L > 280\text{M})$$

[4] Since a run may be terminated or may become unresponsive (UT or Hang outcome type) before step 11, the percentage of runs that require simulation steps 11 and 12 is less than the sum of all erroneous outcome rates presented in Sec. 3.3.3.

Table 3.2. Mixed-mode simulation performance per each step.

Simulation type		Cycles (average)	Performance (cycles/sec.)	Execution time (sec.)
Mixed-mode simulation	Steps 1-2	1M	20K	50
	Steps 3-10	10K	500	20
	Steps 11-12	*L*/2 × 1%	20K	*L*/4M
	Total			70+*L*/4M

3.3. Soft Error Injection Results for Uncore Components

Using the mixed-mode error injection platform, we performed soft error injection runs for uncore components in the OpenSPARC T2 design (Table 3.3). In this chapter, we study soft errors in the L2 cache controller (L2C), the DRAM controller (MCU), the Crossbar interconnect (CCX), and the PCI Express I/O controller (PCIe)[5].

[5] NIU, SIU, and NCU are excluded from this study since RTL simulation of those components requires additional high-level models available only for the Solaris OS on SPARC machines.

Table 3.3. Processor core and uncore components in OpenSPARC T2.

Component	Number of instances	Number of Flip-flops (per instance)	Gate count (per instance)
Processor Core	8	44,288	513,597
L2C	8	31,675	210,540
MCU	4	18,068	155,726
CCX	1	41,521	370,738
PCIe[6]	1	29,022	376,988
NIU	1	135,699	1,297,427
SIU	1	16,908	105,695
NCU	1	17,338	143,374

3.3.1. Flip-flops Targeted for Error Injection

Our soft error injection study excludes flip-flops that are already protected or inactive during normal operation. L2C, MCU, and PCIe have built-in error detection and recovery / error correction, such as ECC and CRC, to address errors inside memory arrays. Flip-flops storing ECC or CRC encoded data are effectively protected. Since a single bit-flip in those flip-flops does not affect application-level behavior (after error correction / recovery), they are excluded from error injection. The inactive flip-flops are dedicated to built-in self-test and redundant arrays to repair defective SRAM cells. For this study, we assume a defect-free chip where these flip-flops are not utilized. Table 3.4 shows the number of flip-flops targeted for error injection in the L2C, MCU, CCX, and PCIe modules.

[6] Because the OpenSPARC T2 distribution does not provide RTL source of the PCI Express controller, we used an industrial implementation of state-of-the-art PCI Express generation 3 controller design to model soft error effects in I/O controllers.

Table 3.4. Number of flip-flops in the targeted uncore components.

Uncore component (number of instances in OpenSPARC T2)	Error injection target flip-flops per instance (% of total flip-flops)	Excluded from error injection	
		Protected	Inactive
L2C (8)	18,369 (58.0%)	8,650 (27.3%)	4,656 (14.7%)
MCU (4)	12,007 (66.4%)	4,782 (26.5%)	1,279 (7.1%)
CCX (1)	41,181 (99.2%)	0 (0%)	340 (0.8%)
PCIe (1)	23,483 (80.9%)	5,539 (19.1%)	0 (0%)

3.3.2. Benchmark Applications

We use a wide range of multi-threaded benchmark applications: 6 SPLASH-2 benchmarks [Woo 95], 9 PARSEC-2.1 benchmarks[7] [Bienia 11], and 3 Phoenix MapReduce benchmarks for shared-memory systems [Yoo 09] (Table 3.5). To fully utilize OpenSPARC T2's 64 hardware threads, we instantiated 64 threads for each benchmark application. For PCIe error injections, we modeled a situation where PCIe I/O is used to transfer the application's input data files. In our benchmark set, 12 applications have input data file as shown in Table 3.5, and they are used for PCIe error injection runs. For each benchmark, we ran more than 40,000 error injection runs

[7] Facesim application is not tested because the input file for simulation is not included in the benchmark suite. Raytrace application from PARSEC is not tested because it produces no output files, and it is not possible to validate the application results.

for each target uncore component. We assume that only one soft error happens for each application run[8].

[8] The interval between flip-flop soft errors is usually much longer compared to the length of the target benchmark applications [Mukherjee 05]. Actual failure rate of the system can be derived by applying technology-dependent soft error rate to the observed application-level outcome rates per injected soft error.

Table 3.5. Benchmark applications for uncore component soft error injection simulations.

Benchmark application		Error-free execution time (cycles)	Input data file size
SPLASH-2	Barnes (barn)	413M	No input file
	Cholesky (chol)	531M	1.7MB
	FFT (fft)	862M	No input file
	LU-contiguous (lu-c)	215M	No input file
	Radix (radi)	120M	No input file
	Raytrace (rayt)	1,005M	4.5MB
PARSEC-2.1	Blackscholes (blsc)	164M	258KB
	Bodytrack (body)	571M	2.5MB
	Ferret (ferr)	763M	4.7MB
	Fluidanimate (flui)	842M	1.3MB
	Freqmine (freq)	353M	8.0MB
	Streamcluster (stre)	695M	No input file
	Swaptions (swap)	591M	No input file
	Vips (vips)	1,003M	7.6MB
	X264 (x264)	881M	2.8MB
Phoenix MapReduce	Linear regression (p-lr)	54M	108MB
	String match (p-sm)	248M	108MB
	Word count (p-wc)	566M	99MB

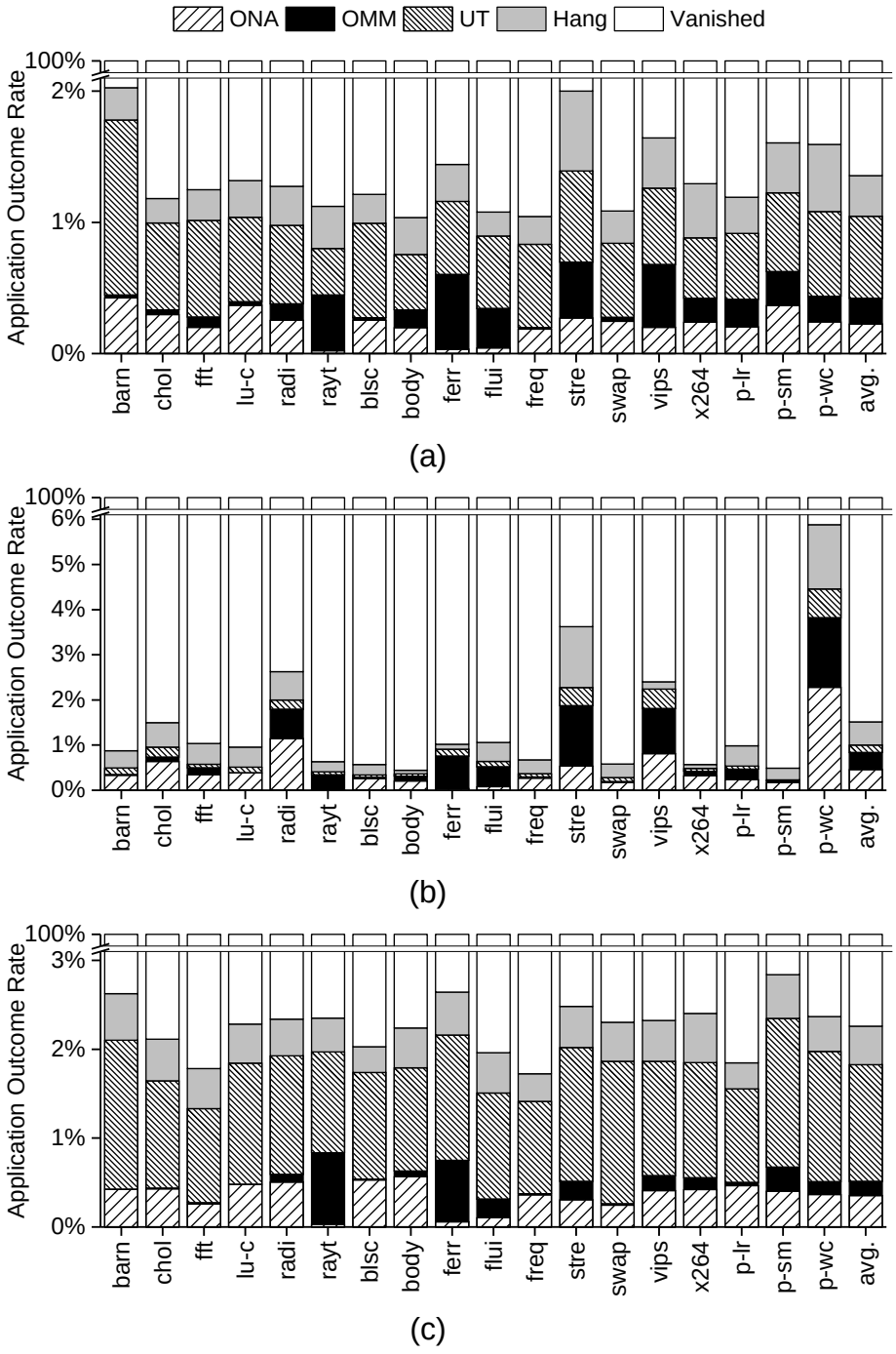
We used the following five outcome categories to classify application-level outcomes (same as Chapter 2): 1) Application Output Not Affected (ONA), 2)

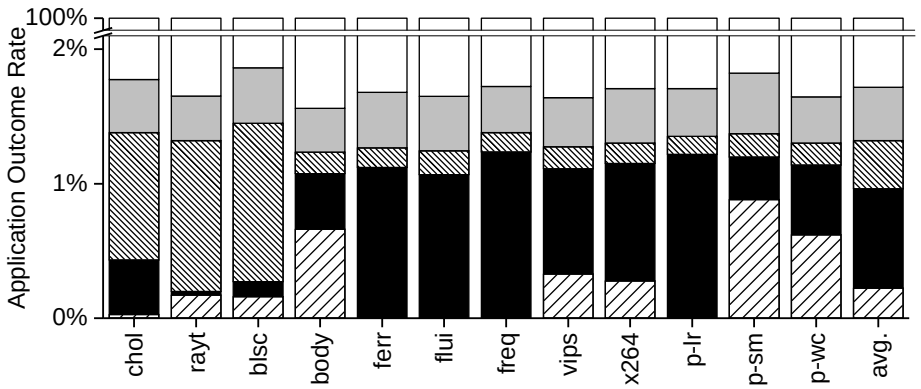
Application Output Mismatch (OMM), 3) Unexpected Termination (UT), 4) Hang, and 5) Vanished.

3.3.3. Application-level Erroneous Outcome Rates

Our soft error simulation results demonstrate that uncore soft errors can have significant impact on the overall chip-level soft error rate. Figure 3.3 shows the observed erroneous outcome rates for each of the uncore components across the benchmark applications and their arithmetic means. For example, in Fig. 3.3a, error injections into L2C for Barnes resulted in 0.42% of ONA, 0.02% of OMM, 1.34% of UT, 0.26% of Hang, and 97.96% of Vanished outcomes.

As expected, most injected soft errors resulted in the Vanished outcome type (over 97% of cases on average). Out of non-Vanished outcomes, UT is the most frequent outcome type for L2C and CCX errors (0.69% on average). However, depending on the application, OMM rates are also significant. For example, the OMM rate for L2C is 0.3% for Fluidanimate and 0.42% for Streamcluster. PCIe error injection results show higher OMM rates (0.89% on average) compared to other components. Since PCIe transfers input data files in our simulations, soft errors in the PCIe likely affect data values. On the other hand, soft errors in other uncore components may corrupt control-related program variables, such as pointers or condition variables that may result in UT or Hang outcomes. Overall, the probability of having an erroneous application outcome (non-Vanished) for a single flip-flop soft error is 1.4%, 1.7%, 2.2%, and 1.7% for L2C, MCU, CCX, and PCIe, respectively.


ONA
OMM
UT
Hang
Vanished
Application Outcome Rate
100%
2%
1%
0%
barn
chol
fft
lu-c
radi
rayt
blsc
body
ferr
flui
freq
stre
swap
vips
x264
p-lr
p-sm
p-wc
avg.
(a)
100%
6%
5%
4%
3%
2%
1%
0%
(b)
100%
3%
2%
1%
0%
(c)



(d)

Figure 3.3. Application-level erroneous outcome rates resulting from error injection for uncore components. (a) L2C. (b) MCU. (c) CCX. (d) PCIe.

The OMM outcome type is a serious reliability concern because, unlike the UT and the Hang outcome types, the user may not be aware that the application resulted in erroneous outputs (unless there are additional mechanisms to verify the correctness of outputs). Figure 3.4 compares the observed OMM rates obtained from our uncore soft error injection runs to the OMM rates of processor core soft errors reported in the literature[9]. The observed OMM rates of uncore soft errors are comparable to that of processor cores, showing that understanding soft error resilience is important for uncore components in the studied OpenSPARC T2 design.

[9] The results are based on injecting one soft error into a single target component (single uncore component or single processor core). The results do not reflect any radiation-hardening techniques or device technologies that have stronger soft error resilience (e.g., SOI [Loveless 11, Oldiges 09]).

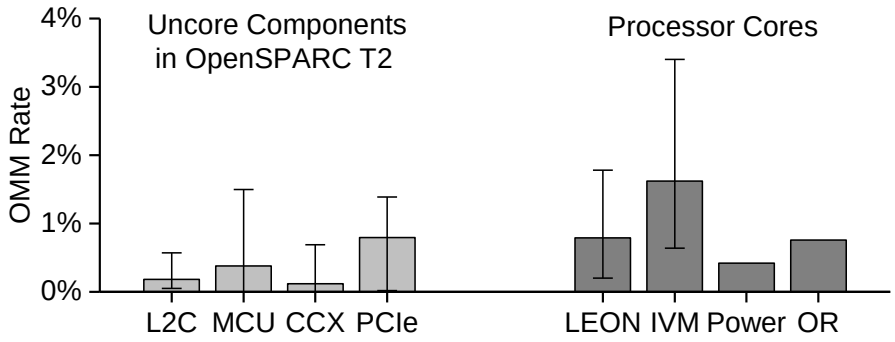


Figure 3.4. OMM rate of uncore components and processor cores (per instance). Error bars are showing the minimum and maximum values observed across benchmark applications. (LEON: LEON3 SPARC [Cho 13], IVM: IVM ALPHA [Cho 13], Power: IBM POWER6 [Sanda 08], and OR: OpenRISC [Meixner 07]).

3.4. Mixed-mode Platform Accuracy

Unlike RTL-only simulations or FPGA-based emulations, where the system is simulated at the flip-flop level all the time, our mixed-mode platform models detailed flip-flop behaviors only during the co-simulation mode. Hence, it is important to quantify the accuracy of our approach.

3.4.1. Warm-up Period of Co-simulation Mode

To show that only a 1,000 cycle warm-up period is enough to restore the microarchitectural states not included in the high-level uncore model (before an error is injected at the flip-flop), we compared the logic value of each microarchitectural state bit of our mixed-mode simulation setup (during co-simulation mode) vs. a simulation setup that runs the RTL co-simulation from the very beginning (i.e., *full-*

co-simulation). In Fig. 3.5, the Y-axis represents the percentage of bits in our mixed-mode setup (during co-simulation mode) that do not match the corresponding bit in the full-co-simulation mode (unless the bit in the full-co-simulation mode is still unknown). The results are averaged over 10,000 runs. After 1,000 cycles into the co-simulation mode, the microarchitectural state of our mixed-mode platform closely matches that of the full-co-simulation (difference less than 0.2%).

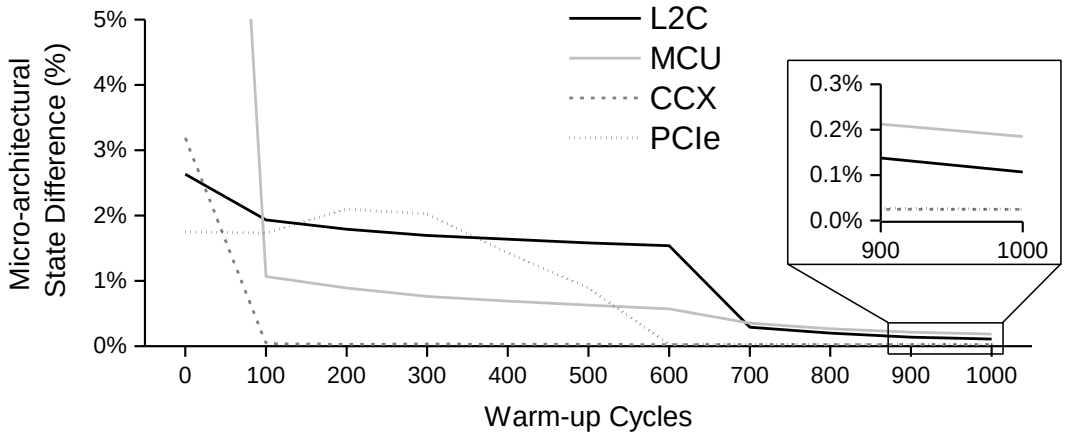


Figure 3.5. Microarchitectural state difference during the warm-up period.

3.4.2. Limited Co-simulation Length

As discussed before, the co-simulation mode terminates early if the outcome of the application run is determined or if only states modeled by high-level uncore models are erroneous. However, in a few cases, errors may persist in uncore microarchitectural states not modeled by high-level uncore models for extended periods of simulation time. For these cases, limiting co-simulation length is a trade-off

between simulation efficiency and accuracy of the obtained results. For our error injection study, only a small subset of soft errors that are injected into a small number of flip-flops result in such situations[10] past 100K cycles of co-simulation. Hence, we limit co-simulation length to 100K cycles. These flip-flops represent 3.7%, 2%, 3.4%, and 3.3% of all flip-flops in L2C, MCU, CCX, and PCIe, respectively (Fig. 3.6). Out of all error injection runs, only 1.8% actually result in situations in which errors in uncore microarchitectural states not modeled by high-level uncore models persist past 100K co-simulation cycles (L2C: 1.8%, MCU: 0.4%, CCX: 1.5% and PCIe: 1.4% of their respective total runs).

Extending the co-simulation length beyond 100K cycles slows down simulation and has diminishing returns in further determining application outcomes (e.g., extending co-simulation cycle limit by 10× to 1M cycles increases the co-simulation time 10-fold, but the percentage of error injection runs for L2C with errors persisting beyond the cycle limit is reduced from 1.8% to 1.4% only). Since these errors might vanish if given more co-simulation cycles, we do not report them as erroneous outcomes in Figs. 3.3 and 3.4. However, one may conservatively choose to protect these flip-flops for error resilient design, as we did in our study of QRR described in Sec. 4.3.

[10] These situations include errors injected into the flip-flops that are not accessed during the co-simulation mode (therefore the mismatches with the corresponding flip-flops in the golden component persist). As discussed in Sec. 3.2.2, persisting flip-flop errors that will eventually be masked (e.g., corrupted data field when the associated valid flag is not set) are not considered as persisting errors that require extended co-simulation length.

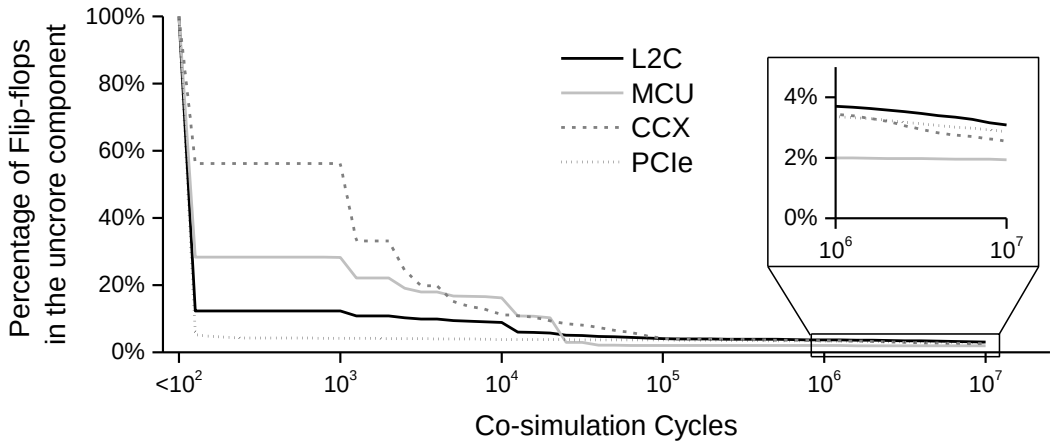


Figure 3.6. Percentage of flip-flops that result in situations in which errors in uncore microarchitectural states not modeled by high-level uncore models persist beyond the given co-simulation cycles.

3.4.3. Application-level Outcomes Accuracy

We compare the observed outcome rates from our mixed-mode platform vs. those obtained from RTL-only simulations. Due to the slow speed of RTL simulators, the comparison is limited to the FFT application with a smaller data set (1M cycles of execution time), running on 4 threads without an OS. ONA and OMM types are categorized into one outcome type because no specific output generation function (e.g., file write) is implemented in this setup. Figure 3.7 compares the observed application-level erroneous outcome rates from the two setups obtained from 40,000 error injection samples each. The observed rates from our mixed-mode platform closely match (0.9-1.1×) those from the RTL-only simulations.

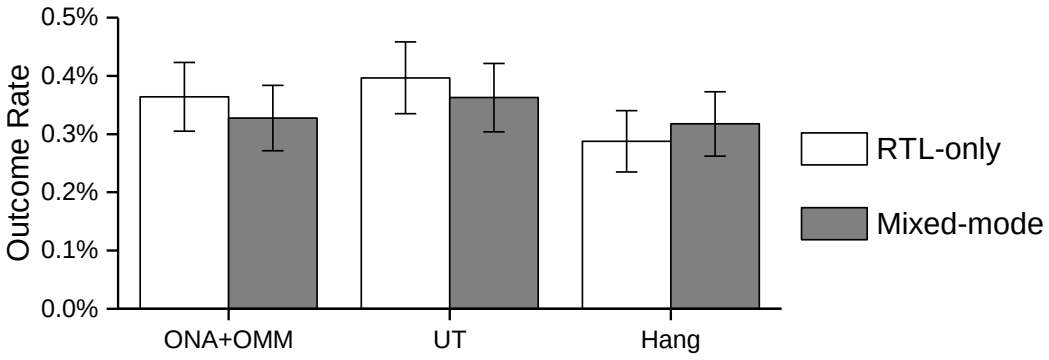


Figure 3.7. Comparison of observed erroneous outcome rates from RTL-only simulations vs. those from our mixed-mode platform. Error bars represent the 95% confidence intervals.

3.5. Conclusion

Studying the application-level effects of uncore soft errors in large-scale SoCs is important but difficult. Our new mixed-mode simulation platform enables us to accurately and effectively model uncore soft errors while achieving 20,000-fold speedup compared to RTL simulations. This platform enabled us to characterize, for the first time, system-level effects of soft errors in various uncore components of a large and industrial-grade multi-core SoC.

Our results show that uncore soft errors can have significant impact on the overall reliability of for the studied OpenSPARC T2 multi-core SoC. Hence, resilience techniques to overcome uncore soft errors are required. In the next chapter, we discuss the challenges for the uncore soft error resilience and introduce our new uncore soft error resilience technique.

Chapter 4

Soft Error Resilience for Uncore Components

4.1. Introduction

As we discussed in Chapter 3, soft errors in uncore components can have significant impact on system-level reliability. Therefore, it is important to provide efficient soft error resilience mechanisms to uncore components to ensure the robust operations of the entire system, including processor cores and uncore components.

In this chapter, we make the following contributions:

1. We show that traditional system-level checkpoint recovery techniques that generally target processor cores are inadequate for uncore components.
2. We present a new soft error recovery technique called Quick Replay Recovery (QRR) for uncore components belonging to the memory subsystem. We

demonstrate the effectiveness of QRR for the L2 cache controller and the DRAM controller in the OpenSPARC T2 design. QRR results in 100× improvement (i.e., reduction) of the probability that an application run fails to produce correct results due to soft errors in uncore components belonging to the memory subsystem; the corresponding chip-level area and power impact for all L2C and MCU instances are 1.44% and 2.49%, respectively.

4.2. System-level Checkpoint Recovery Challenges

Many error resilience solutions depend on system-level checkpoint recovery techniques to revert the system to an error-free state upon error detection [Elnozahy 02]. One major challenge for ensuring correct recovery is the *output commit problem* that may incur long delays for system outputs. Since rollback recovery may not be able to invalidate committed outputs to the outside world[1], such as network packets or human interactions, outputs should be committed only when it is guaranteed that the system will not roll back to a state before the outputs were produced [Elnozahy 02, Nakano 06]. To avoid such long output delays, two conditions must be satisfied: 1) errors must be detected quickly and 2) the recovery operation should not revert the system to a very old state during rollback to an error-free state (i.e., *rollback distance* should be short).

[1] Outputs destined beyond the boundary of recovery (*sphere of recovery*). For example, if the recovery does not include disks, a write to disk may not be nullified. A mechanism with a broader sphere of recovery may mitigate the output commit problem, but the associated overheads can be high. For example, it may take up to half an hour to create a checkpoint for petascale systems [Cappello 09].

4.2.1. Long Error Detection Latency of Uncore Soft Errors

Having long *error detection latencies*, i.e., the time elapsed from the cycle the soft error affects the uncore component[2] to the cycle the error is detected by an error detection technique, may cause the system to roll back to a very old state in order to revert the system to an error-free state. Error detection techniques at the software and processor architecture levels, such as EDDI [Oh 02] and RMT [Mukherjee 08], can detect uncore errors only after a processor core sees an erroneous output from the uncore component. Therefore, the shortest error detection latency for such techniques is longer than the *error propagation latency to processor cores*, i.e., the duration from the cycle when a soft error affects an uncore component[3] until the cycle when uncore component produces an erroneous output to the processor cores.

For soft errors injected in the uncore components associated with the memory subsystem (L2C, MCU, and CCX) of OpenSPARC T2, we observed very long error propagation latencies (Fig. 4.1). For example, soft errors in L2C take 36 million cycles to propagate to processor cores on average. For processor cores, in contrast, errors can be detected quickly within a short amount of time [Maniatakos 11a, Smolens 04]. Proactively loading and comparing memory values from uncore components can reduce error propagation and detection latencies [Lin 14, 15]. These techniques have been successfully used for validation purposes. The use of hardware support for such

[2] The cycle when the soft error causes a bit-flip in the uncore component.

[3] In this section, we use a conservative approach to quantify the resulting rollback distance. As long as a recent checkpoint is error-free, a soft error detection may not result in a rollback to a very old state even if the detection occurs a long duration after the actual bit-flip in the flip-flop. Therefore, we measure the error propagation latency (≤ error detection latency) from the cycle when the flip-flop error eventually results in a corruption in the states that are included in the checkpoint (e.g., software-visible state of the main memory for the uncore components belonging to the memory subsystem).

approaches [Lin 15] can be promising for soft error detection and require further experimentation and analysis.

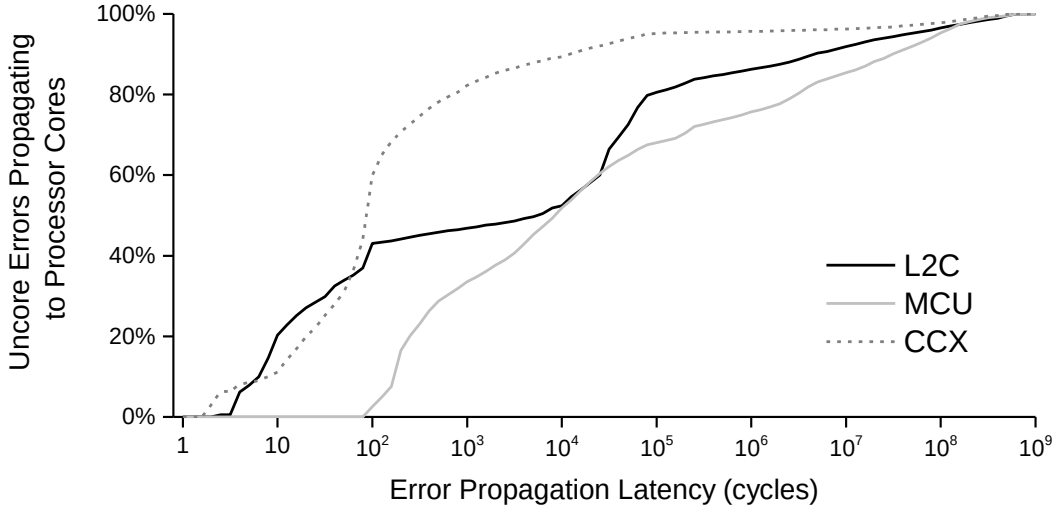


Figure 4.1. Cumulative distribution of uncore error propagation latencies to processor cores.

4.2.2. Long Rollback Distance for Uncore Soft Errors

To ensure short rollback distance, the checkpointing mechanism has to create checkpoints frequently (short checkpoint interval). To frequently create checkpoints, the data size of each checkpoint has to be kept small due to the limited checkpoint storage size and bandwidth. To achieve small checkpoint data size, *incremental checkpointing* techniques are used [Prvulovic 02, Sorin 02]. They reduce the data size

of each checkpoint by saving logs of memory locations [4] modified by processor cores between two checkpoints.

For soft errors in uncore components, however, such techniques may not be adequate. For example, suppose that processor cores modified memory contents in the address range [*X*-*Y*] (and, hence, only those memory contents were included in an incremental checkpoint). However, a soft error in L2C might corrupt the content of memory address *Z* which is outside the range [*X*-*Y*] (due to an address-related error). In such a case, the recovery mechanism must roll back to an older state with an error-free log on address *Z*.

The required roll back distance to recover from corrupted values in an arbitrary memory location is determined by when a processor core last modified that memory location.

Figure 4.2 shows the cumulative distribution of required rollback distances resulting from soft errors in L2C and MCU. To cover more than 99% of soft errors resulting in memory corruptions, the required rollback distance can be longer than 400M cycles.

[4] Other architectural states, such as register values, have much smaller size compared to the main memory state, and may not require incremental checkpointing.

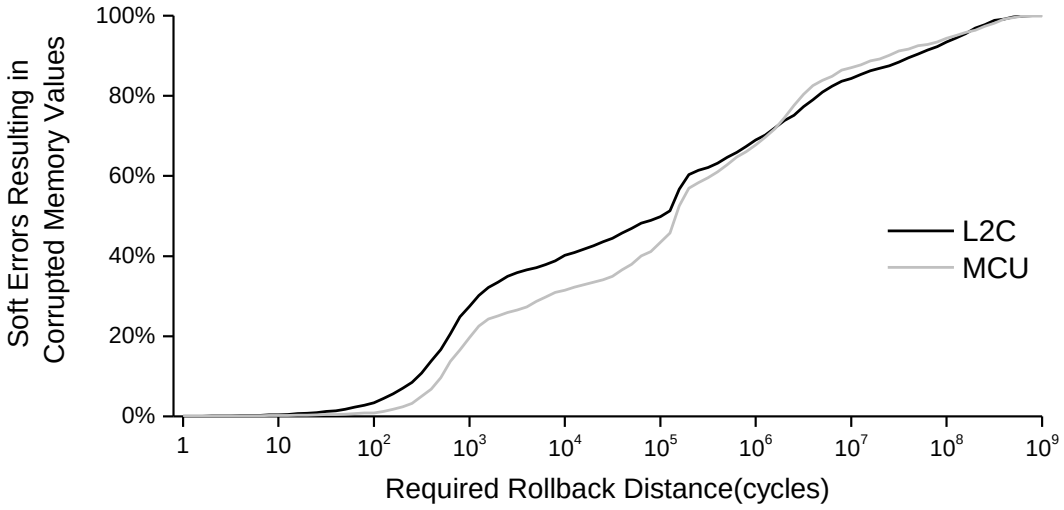


Figure 4.2. Cumulative distribution of required rollback distance resulting from soft errors in L2C and MCU.

4.3. Uncore Soft Error Resilience Using Quick Replay Recovery

Uncore soft error resilience can be achieved by utilizing radiation-hardened flip-flops [Lilja 13, Mitra 05], but the associated costs may not be optimal (Sec. 4.3.5). Logic parity [Mitra 00] can detect errors with very short error detection latency; combined with an efficient recovery technique, logic parity can provide a low-cost error resilience solution. For processor cores, efficient error recovery techniques exist (e.g., by flushing instructions [Ando 03, Mukherjee 08], or by using instruction-level retry [Meaney 05]). For uncore components, such mechanisms are inadequate due to the following reasons:

1. As discussed in Sec. 3.2.1, uncore components process request packets from processor cores. Those request packets need to be recreated for recovery. An uncore component may not be able to regenerate request packets by itself.

2. Requesting processor cores to resend request packets may not always be possible since processor cores may not store information about request packets being processed by uncore components. For example, OpenSPARC T2 processor cores retain request packets only until L2C sends corresponding return packets. However, L2C may continue to process a request even after sending the return packet to the processor core. For example, if a request results in a store miss, L2C may spend hundreds of cycles to fetch a cache line even after sending the return packet. In this case, the uncore operation may be affected by a soft error even after the processor removes the request packet (upon receipt of the return packet).
3. Reverting processor cores, along with the erroneous uncore component, may result in cascaded rollbacks since each uncore component can interact with multiple processor cores and/or uncore components. For example, rolling back a processor core might require rolling back the uncore components the processor core interacted with, such as other instances of L2C. This, in turn, might require rolling back other processor cores that interacted with those uncore components.

To overcome these challenges, we present a new technique called *Quick Replay Recovery* (*QRR*) targeting uncore components (Fig. 4.3). QRR handles soft errors without engaging processor cores during recovery. It is applicable for uncore components that satisfy the following properties:

1. Executing requests multiple times in the same order does not change the outcome. For example, this property is maintained in storage components such as memory where duplicated operations in the same order do not change the

outcome. (For a detailed discussion regarding this property in the presence of requests accessing the same address, please refer to Sec. 4.3.3).

2. The uncore component should be able to resume its operation upon reset of its flip-flop contents. For flip-flop contents that should not be reset, such as flip-flops used for configuration bits (e.g., cache disable bit in L2C), radiation-hardening can be selectively used to protect those flip-flops (fewer than 3% for L2C and MCU) from soft errors.

In this study, QRR works in conjunction with logic parity-based error detection (other error detection techniques with very short error detection latencies are also possible). It provides the following functionality:

1. Record request packets using a record table in the QRR controller. Packets are stored in that table when a new request packet is sent to the uncore component, and deleted from the table when the associated operation is completed by the uncore component (Details in Sec. 4.3.1). Flip-flops in the QRR controller are protected using radiation hardening.
2. When logic parity detects an error, the QRR controller performs recovery operation by resending the request packets in the record table to the uncore component (Details in Sec. 4.3.2).

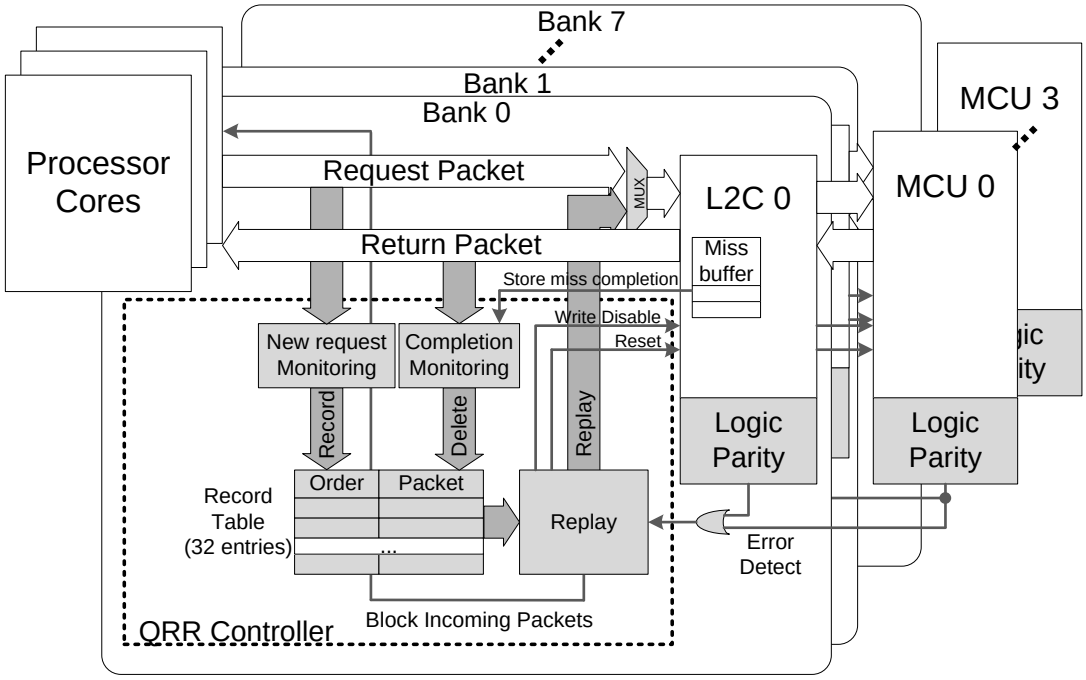


Figure 4.3. QRR for L2C and MCU. QRR components are shaded.

We evaluate QRR for the L2C and MCU modules for which traditional checkpoint recovery techniques are inadequate (Sec. 4.2). Because MCU receives access requests through L2C only (e.g., cache line fill, eviction, or non-cached direct DRAM access), recording and replaying L2C requests effectively covers MCU requests as well[5]. QRR incurs a small performance impact during recovery. For L2C, in the worst case when every replayed packet results in the longest operation (L2 cache load miss), the recovery takes fewer than 5,000 cycles.

[5] Since an MCU instance operates with two L2C instances in OpenSPARC T2, soft error detection in an MCU invokes recovery operation of two QRR controllers in the two L2C instances.

4.3.1. QRR Normal Operation

During normal operation, the QRR controller keeps track of request packets that are being processed in the uncore component using its record table. QRR for an L2C instance maintains a total ordering of all incomplete requests to that instance based on their arrival order. This is a stricter ordering than the original design, which only needs to maintain the arrival ordering between requests to the same cache line in order to preserve the required SPARC total store ordering (TSO) [OpenSPARC]. Since each L2C and MCU instance exclusively serves disjoint memory address ranges, maintaining ordering at each L2C instance (bank) is sufficient (without affecting requests being processed by other instances).

When requests are completed without errors, they no longer need to be stored by the QRR controller. A completion of a request is determined by monitoring return packets to the processor cores. For uncore requests that require post processing even after the return packet, additional monitoring may be required. For our QRR implementation targets (L2C and MCU), the only return packet type requiring additional monitoring is a store miss (described as an example in Sec. 4.3). In this case, the QRR controller waits until the cache miss handling logic (*Miss Buffer*) in L2C completes the operation before deleting the corresponding entry.

4.3.2. QRR Replay Recovery Operation

When logic parity detects an error, QRR first disables write enable signals to data arrays (e.g., L2 cache tag, data, and DRAM) and valid signals of data ports

connected to processor cores or other uncore components to prevent the error from corrupting those states and propagating to other components.

Propagating the parity error detection signal (*individual error signal*) to the QRR controller and invoking the recovery operation may take multiple cycles because signals from multiple parity detectors have to be aggregated. If a (detected) flip-flop error propagates to a data array or to another component within a few cycles vs. the number of cycles required to propagate the aggregated error signal to the QRR controller, then the soft error might corrupt the corresponding data array or the connected component before the recovery operation is invoked. This creates a non-zero chance of corrupt outputs being produced by the SoC. In our current implementation, we managed this issue by manually inspecting cases where such situations might arise, and fixed the issues by routing individual error signals to corresponding data arrays or other components before the error signals are fully aggregated into the input for the QRR controller. These manually routed signals disable write enable signals and valid signals before the flip-flop error propagates to those arrays or components.

The next step is to assert the reset signal of the uncore component to clear its flip-flop values. Accepting new request packets from processor cores is postponed until recovery is completed. After reset, the QRR controller sends recorded packets to the uncore component in the recorded order until all recorded incomplete request packets are replayed. After the replay completes, the uncore component resumes normal operation by starting to accept new request packets from processor cores.

QRR can successfully recover errors for the following reasons:

1. For L2C and MCU, executing incomplete request packets again (replay) does not change the outcome. As long as multiple concurrent requests do not access the same address, replaying requests in a given order results in the same outcome. For example, executing requests "*read from X*", "*write A to Y*", and "*read from Z*" multiple times have the same resulting effects. If there are dependencies (i.e., multiple requests that access the same address) between concurrent requests, executing requests multiple time may result in a different outcome (e.g., "*read from X*" and "*write A to X*" when the original value of *X* is not *A*). For requests that exhibit dependencies, as discussed in Sec. 4.3.1, L2C in OpenSPARC is originally designed not to begin the execution of the following request until the previous one completes (i.e., only one of the requests are executed at a time). Therefore, the replay by QRR does not result in a situation where multiple dependent requests are executed multiple times. For example, suppose that requests *R1*, *R2*, and *R3* have dependencies to each other, and L2C executes the packets in *R1*→*R2*→*R3* order. If a soft error is detected when L2C executes *R2*, the replay by QRR results in a re-execution of *R2* only for the following reasons: 1) The execution of *R1* is already completed at that point (removed from the record table) and not included in the replay. 2) The execution of *R3* has not been started yet and the actual execution of *R3* happens only after the replay (no multiple execution of *R3*).
2. By enforcing a stricter ordering between recorded requests (vs. the default memory ordering of the target uncore component), requests replayed by QRR do not violate the memory access order of the original requests.

3. A detected soft error does not change the outcome of replayed operations since the erroneous flip-flop values are reset by the QRR controller, the contents of the SRAM and DRAM arrays are preserved, and data signals to other components are invalidated (except for the corner case situation related to the error signal propagation discussed in Sec. 4.3.2).

4.3.3. QRR Results

We implemented QRR for the L2C and MCU modules of OpenSPARC T2, and evaluated its effectiveness using the mixed-mode platform discussed in Chapter 3.

To minimize the cost of parity-based error detection, we selectively used radiation hardening for the following flip-flops:

1. Flip-flops with timing slack shorter than the path delay of the XOR tree used to calculate a parity bit. In such a case, logic parity may not be a cost-effective solution since it is not possible to place the XOR tree without slowing down the clock or using additional flip-flops to split the XOR tree over multiple clock cycles. 1,650 flip-flops of L2C (9%), 36 flip-flops of MCU (0.3%) belong to this category.
2. Configuration flip-flops where reset and replay may fail to restore the required flip-flop values. These flip-flops are excluded from reset. 55 flip-flops of L2C (0.3%), 309 flip-flops of MCU (2.5%) belong to this category.
3. Flip-flops in the QRR controller. 812 flip-flops per instance (~3% of the flip-flops in L2C and MCU) belong to this category. Since the flip-flops in the QRR

controller are hardened, we did not protect the tables in the QRR controller (assuming single soft errors).

The rest of the flip-flops in the uncore components are protected by logic parity and QRR.

From simulations using the same set of applications as in Sec. 3.3.2, QRR successfully recovered from all errors injected into the flip-flops covered by logic parity for over 400,000 error injection runs for L2C and MCU.

4.3.3.1. Selective Flip-flop Protection Using QRR

To maximize the cost-effectiveness of a soft error resilience solution, not every flip-flop in the design needs to be protected. We evaluate the resulting *error resilience improvement* of QRR by comparing the sum of observed rates[6] of non-Vanished outcome types[7] (*non-Vanished outcome rate*).

$$Error\ resilience\ improvement = \frac{Non\text{-}Vanished\ outcome\ rate\ of\ the\ original\ design}{Non\text{-}Vanished\ outcome\ rate\ with\ QRR}$$[8]

[6] Since QRR does not introduce execution time overhead in error-free mode (and introduces short recovery upon error detection), QRR does not change the error-free execution time of a given application. Therefore, we assume that the probability of having a soft error during the execution of a given application is the same for the original design and the design with QRR.

[7] ONA, OMM, UT, and Hang.

[8] When calculating the error resilience improvement for QRR, flip-flops protected using logic parity error detection (and QRR recovery) are considered to have zero probability of having a non-Vanished outcome (Sec. 4.3.5). Assuming 1,000× soft error rate reduction of radiation-hardened flip-flops [Lilja 13], flip-flops protected using radiation hardening are considered to have 0.001× probability of having a non-Vanished outcome compared to that of the corresponding flip-flip in the original (i.e., without radiation-hardened flip-flops) design.

For example, if the non-Vanished outcome rate of the original design is 1% and the design with QRR reduces the rate to 0.1%, the resulting error resilience improvement is 10×.

Flip-flops have varying degrees of *vulnerability* (i.e., the likelihood that a soft error in the flip-flop cause non-Vanished outcome types). To meet the desired level of error resilience improvement (*error resilient improvement goal*), we can selectively provide soft error resilience solutions to protect flip-flops with higher vulnerability first to minimize the associated overheads. Table 4.1 shows the error resilience improvement goals and the percentage of flip-flops that need to be protected if we prioritize flip-flops based on their vulnerabilities[9]. For example, to achieve 5× error reilience improvement, 25.10% of flip-flops with the highest vulnerabilities in L2C and MCU need to be protected.

Table 4.2 presents the area and power overheads of QRR for the corresponding error resilience improvement goals. The area and power overheads are obtained after synthesis and place-and-route[10]. Table 4.2 also compares the overheads of QRR to those of the hardening-only mechanism that protects the same set of flip-flops using radiation-hardened flip-flops. Compared to the hardening-only mechanism, QRR

[9] In this section, we evaluated the vulnerability of a flip-flop based on the error injection results across all benchmark applications in Sec. 3.3. However, the vulnerability of a flip-flop may vary depending on the application (*benchmark dependence*). A soft error resilience solution optimized for a specific set of applications may not result in the best error resilience for applications not included in the set. To estimate the resulting error resilience for an arbitrary application, one may divide the benchmark applications into two groups, one group of applications for selecting flip-flops to protect (referred to as the *training set*) and another (disjoint) group of applications for evaluating the error resilience (referred to as the *evaluation set*).

[10] The area overhead is obtained using the Synopsys Design Compiler and a commercial 28nm technology library. The power overhead is calculated using the Synopsys PrimeTime and application execution traces. Chip-level overhead is estimated based on published data in related OpenSPARC T2 studies [Jung 14, Li 13].

achieves up to 20% and 13.6% lower area and power overhead, respectively, for 100× error resilience improvement. When the number of protected flip-flops is small, QRR does not show significant improvements or even shows worse result than the hardening-only approach.

Table 4.1. Error resilience improvement goals and the required portions of flip-flops that need to be protected.

Error resilience improvement goal	Flip-flops in L2C and MCU that need protection (%)
2×	10.91%
5×	25.10%
10×	31.33%
20×	35.99%
50×	44.71%
100×	57.11%
200×	66.75%
500×	71.95%
1000×	79.28%

Table 4.2. QRR overhead comparison. The percentages are overheads for L2C and MCU (the percentages in parenthesis are chip-level overheads).

Error resilience improvement goal	Area overhead % (chip-level %)			Power overhead% (chip-level %)		
	QRR	Hardening only	Improvement	QRR	Hardening only	Improvement
2×	3.28% (0.24%)	0.17% (0.01%)	-	1.45% (0.19%)	0.44% (0.06%)	-
5×	3.67% (0.27%)	0.44% (0.03%)	-	2.17% (0.28%)	0.59% (0.08%)	-
10×	4.23% (0.31%)	1.13% (0.08%)	-	2.27% (0.29%)	0.99% (0.13%)	-
20×	5.59% (0.40%)	2.67% (0.19%)	-	3.03% (0.39%)	4.99% (0.64%)	-
50×	7.27% (0.53%)	4.02% (0.29%)	-	4.23% (0.54%)	4.21% (0.54%)	-
100×	19.97% (1.44%)	23.55% (1.70%)	15.18%	19.35% (2.49%)	21.56% (2.77%)	10.27%
200×	22.69% (1.64%)	28.01% (2.03%)	18.97%	23.01% (2.96%)	24.56% (3.16%)	6.32%
500×	23.96% (1.73%)	30.32% (2.19%)	20.96%	25.80% (3.32%)	28.53% (3.66%)	9.54%
1000×	25.71% (1.86%)	32.35% (2.34%)	20.52%	26.77% (3.44%)	30.98% (3.98%)	13.59%

4.3.3.2. QRR Overheads Breakdown

To show the details of the area and power overheads, Table 4.3 presents the breakdown of the overheads associated with the QRR implementation for L2C and

MCU. For this breakdown, QRR protects all flip-flops (i.e., flip-flops subject to error injection in Table 3.4) in L2C and MCU. In this case, the majority of the overheads come from logic parity or flip-flop hardening. The total area and power overheads of QRR are 31.33% and 35.2% at each uncore component level (2.27% and 4.52% at chip-level for all L2C and MCU instances).

Table 4.3. QRR area and power overhead breakdown for L2C and MCU. Flip-flops in the QRR controller are protected using radiation-hardening.

Overhead	QRR				Hardening only (chip-level)
	Logic parity	Flip-flop hardening	QRR controller and record table	Total (chip-level)	
Area	20.7%	4.83%	5.8%	31.33% (2.27%)	42.1% (3.05%)
Power	25.0%	6.3%	3.9%	35.2% (4.52%)	40.7% (5.22%)

4.4. Conclusion

Our results show that uncore soft errors can have significant impact on the overall reliability of for the studied OpenSPARC T2 multi-core SoC. Hence, resilience techniques to overcome uncore soft errors are required. However, uncore soft errors pose several challenges for traditional system-level checkpointing techniques that are generally effective for processor cores. Our **Q**uick **R**eplay **R**ecovery approach overcomes these challenges for uncore components in the memory subsystem of OpenSPARC T2. We demonstrate the effectiveness of QRR for L2C and MCU in the

OpenSPARC T2 design. QRR achieves 100× error resilience improvement with the chip-level area and power impact of 1.44% and 2.49%, respectively.

Chapter 5

Concluding Remarks

In this dissertation, we quantify the inaccuracies associated with existing soft error injection techniques and analyze the sources of such inaccuracies. This analysis shows that widely-used soft error injection techniques can lead to highly inaccurate error injection results when evaluating the system-level reliability in the presence of soft errors. To avoid such inaccuracies and obtain high simulation throughputs for large-scale designs, we created a new mixed-mode simulation platform that simulates soft errors in uncore components. Using this platform, we demonstrate that soft errors in uncore components have significant reliability impact, comparable to soft errors in processor core components. This indicates that it is important to study system-level effects arising from soft errors in all components in the system design, not limited to soft errors in processor cores, to achieve system-wide soft error resilience.

We also analyze the challenges of soft error resilience for uncore components using traditional system-level checkpoint recovery. Due to the long error detection latencies associated with soft errors in uncore components, system-level checkpoint

recovery techniques may induce long delays when the system produces outputs to the outside world, known as the output commit problem. For uncore components belong to the memory subsystem, our QRR soft error recovery technique can avoid such challenges while achieving high degrees of soft error resilience at low area and power overheads.

The following research directions can further extend the results of this dissertation:

1) While we focused on soft errors in this dissertation, future work must address other sources of errors as well, e.g., errors induced by manufacturing and environmental variations, early-life failures, and circuit aging.
2) Future research directions should explore cross-layer error resilience techniques (spanning circuit, logic, architecture, software, and application layers) to systematically derive optimized error resilience solutions for a broad range of components, including processor cores, uncore components, and accelerators.

Publications

1. H. Cho, C.-Y. Cher, T. Shepherd, and S. Mitra, “Understanding Soft Errors in Uncore Components,” *Proc. Design Automation Conf.,* 2015.

2. S. Mitra, P. Bose, E. Cheng, H. Cho, R. Joshi, Y. M. Kim, C. Lefurgy, Y. Li, K. Rodbell, K. Skadron, J. Stathis and L. Szafaryn, “The Resilience Wall: Cross-Layer Solution Strategies,” *Proc. IEEE Intl. Symp. VLSI Technology, Systems and Applications and IEEE Intl. Symp. VLSI Design, Automation and Test,* 2014 (invited).

3. S. Mirkhani, H. Cho, S. Mitra, and J. A. Abraham, “Rethinking Error Injection for Effective Resilience,” *Proc. IEEE Asia and South Pacific Design Automation Conf.,* pp. 390–393, 2014 (invited).

4. H. Cho, S. Mirkhani, C.-Y. Cher, J. A. Abraham and S. Mitra, “Quantitative Evaluation of Soft Error Injection Techniques for Robust System Design,” *Proc. Design Automation Conf.,* 2013.

5. H. Cho, L. Leem, and S. Mitra, “Error Resilient System Architecture for Probabilistic Applications,” *IEEE Trans. Comput.-Aided Des. Integr. Circuits and Syst.,* vol. 31, no. 4, pp. 546–558, Apr. 2012.

6. S. Mitra, H. Cho, T. Hong, Y. Kim, H. Lee, L. Leem, Y. Li, D. Lin, E. Mintarno, S. Park, N. Patil, H. Wei and J. Zhang, “Robust System Design,” *IPSJ Trans. System LSI Design Methodology,* 2011 (Invited)

7. L. Leem, H. Cho, J. Bau, Q. A. Jacobson, S. Mitra, "Error Resilient System Architecture for Probabilistic Applications," *Proc. Design, Automation and Test in Europe,* pp. 1560–1565, 2010.

8. L. Leem, H. Cho, H.-H. Lee, Y. M. Kim, Y. Li, S. Mitra, "Cross-layer error resilience for robust systems," *Proc. Intl. Conf. Computer-Aided Design,* pp. 177–180, 2010 (invited).

References

[Ando 03] H. Ando *et al.*, "A 1.3 GHz Fifth Generation SPARC64 Microprocessor ," *Proc. Intl. Solid-State Ciruits Conf.,* pp. 702–705, 2003.

[Arlat 03] J. Arlat *et al.*, "Comparison of Physical and Software-Implemented Fault Injection Techniques," *IEEE Trans. Computers,* vol. 52, no. 9, pp. 1115–1133, Sept. 2003.

[Asaad 12] S. Asaad *et al.*, "A Cycle-accurate, Cycle-reproducible multi-FPGA System for Accelerating Multi-core Processor Simulation," *Proc. Intl. Symp. Field Programmable Gate Arrays,* pp. 153–162, 2012.

[Bailan 10] O. Bailan *et al.,* "Verification of soft error detection mechanism through fault injection on hardware emulation platform," *Proc. Intl. Conf. Dependable Systems and Networks Workshops*, pp. 113–118, 2010.

[Bender 08] C. Bender *et al.*, "Soft-error resilience of the IBM POWER6 processor input/output subsystem," *IBM Journal of Research and Development*, vol. 52, no. 3, pp. 285–292, May 2008.

[Benini 03] L. Benini *et al.*, "SystemC Co-simulation and Emulation of Multiprocessor SoC Designs," *IEEE Computer,* vol. 36, no. 4, pp. 53–59, Apr. 2003.

[Bernick 05] D. Bernick *et al.*, "NonStop® Advanced Architecture," *Proc. Intl. Conf. Dependable Systems and Networks*, pp. 12–21, 2005.

[Bienia 11] C. Bienia, "*Benchmarking Modern Multiprocessors*," *Ph.D. Dissertation*, Princeton University, Princeton, NJ, USA, 2011.

[Borkar 07] S. Borkar, N. P. Jouppi, and P. Stenstrom, “Microprocessors in the Era of Terascale Integration,” *Proc. Design, Automation and Test in Europe*, pp. 237–242, 2007.

[Borkar 11] S. Borkar and A. A. Chien, “The Future of Microprocessors,” *Commun. ACM*, vol. 54, no. 5, pp. 67–77, May 2011.

[Bottoni 14] C. Bottoni *et al.*, “Heavy ions test result on a 65nm Sparc-V8 radiation-hard microprocessor,” *Proc. IEEE Intl. Reliability Physics Symp.* pp. 5F.5.1-5F.5.6, June 2014.

[Cappello 09] F. Cappello, “Fault Tolerance in Petascale/ Exascale Systems: Current Knowledge, Challenges and Research Opportunities,” *Intl. Journal of High Performance Computing Applications*, vol.23, no.3, pp. 212–226, Aug. 2009.

[Chen 06] G. Chen *et al.*, “Object Duplication for Improving Reliability,” *Proc. Asia and South Pacific Design Automation Conf.*, pp. 140–145, 2006.

[Chen 08] D. Chen *et al.*, “Error Behavior Comparison of Multiple Computing Systems: A Case Study Using Linux on Pentium, Solaris on SPARC, and AIX on POWER,” *Proc. IEEE Pac. Rim Intl. Symp. Dependable Computing*, pp. 339–346, 2008.

[Cho 13] H. Cho *et al.*, “Quantitative Evaluation of Soft Error Injection Techniques for Robust System Design,” *Proc. Design Automation Conf.*, 2013.

[Choi 90] G. S. Choi, R. K. Iyer, and V. A. Carreno, “Simulated Fault Injection: A Methodology to Evaluate Fault Tolerant Microprocessor Architectures,” *IEEE Trans. Reliability*, vol. 39, no. 4, pp. 486–491, Oct. 1990.

[Davis 09] J. D. Davis, C. P. Thacker, and C. Chang, "BEE3: Revitalizing Computer Architecture Research," *Microsoft Tech. Rep.* MSR-TR-2009-45, 2009.

[DeHon 10] A. DeHon, H. M. Quinn, and N. P. Carter, "Vision for Cross-Layer Optimization to Address the Dual Challenges of Energy and Reliability," *Proc. Design, Automation and Test in Europe,* pp. 1017–1022, 2010.

[Ejlali 03] A. Ejlali *et al.,* "A Hybrid Fault Injection Approach Based on Simulation and Emulation Co-operation," *Proc. Intl. Conf. Dependable Systems and Networks,* pp. 479–488, 2003.

[Elnozahy 02] E. N. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson, "A survey of rollback-recovery protocols in message-passing systems," *ACM Comput. Surv.*, vol. 34, no. 3, pp. 375–408, Sept. 2002.

[Feng 10] S. Feng, S. Gupta, A. Ansari, and S. Mahlke, "Shoestring: Probabilistic Soft Error Reliability on the Cheap," *Proc. Intl. Conf. Architectural Support for Programming Languages and Operating Systems,* pp. 385–396, 2010.

[Fleming 86] P. J. Fleming and J. J. Wallace, "How not to lie with statistics: The correct way to summarize benchmark results," *Commun. ACM,* vol. 29, no. 3, pp. 218–221, Mar. 1986.

[Gem5] "The gem5 Simulator System," http://www.m5sim.org

[Goswami 97] K. K. Goswami, R. K. Iyer, and L. Young, "DEPEND: A Simulation-Based Environment for System Level Dependability Analysis," *IEEE Trans. Computers,* vol. 46, no. 1, pp. 60–74, Jan. 1997.

[Graham 09] D. Graham, P. Strid, S. Roy, and F. Rodriguez, “A low-tech solution to avoid the severe impact of transient errors on the IP interconnect,” *Proc. Intl. Conf. Dependable Systems and Networks*, pp. 478–483, 2009.

[Gu 04] W. Gu, Z. Kalbarczyk, and R. K. Iyer, “Error Sensitivity of the Linux Kernel Executing on PowerPC G4 and Pentium 4 Processors,” *Proc. Intl. Conf. Dependable Systems and Networks*, pp. 887–896, 2004.

[Gupta 12] V. Gupta *et al.*, “The Forgotten 'Uncore': On the Energy-efficiency of Heterogeneous Cores,” *Proc. USENIX Annual Technical Conf.*, 2012.

[Hauck 07] S. Hauck and A. DeHon, “Reconfigurable computing: the theory and practice of FPGA-based computation,” Morgan Kaufmann, 2007.

[Howard 10] J. Howard *et al*., “A 48-Core IA-32 Message-Passing Processor with DVFS in 45nm CMOS,” *Proc. IEEE Intl. Solid-State Circuits Conf.*, pp. 108–109, 2010.

[Jung 14] M. Jung *et al.*, “On Enhancing Power Benefits in 3D ICs: Block Folding and Bonding Styles Perspective,” *Proc. Design Automation Conf.*, 2014.

[Kalbarczyk 99] Z. Kalbarczyk *et al*., “Hierarchical Simulation Approach to Accurate Fault Modeling for System Dependability Evaluation,” *IEEE Trans. Software Engineering*, vol. 25, no. 5, pp. 619–632, Sept.–Oct. 1999.

[Kanawati 93] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham, “EMAX: An Automatic Extractor of High-Level Error Models,” *Proc. AIAA Computing Aerospace Conf.*, pp. 1297–1306, 1993.

[Kim 07] J. Kim *et al.*, “Multi-bit Error Tolerant Caches Using Two-Dimensional Error Coding,” *Proc. Intl. Symp. Microarchitecture*, pp. 197–209, 2007.

[KleinOsowski 02] AJ KleinOsowski, and D. J. Lilja, "MinneSPEC: A New SPEC Benchmark Workload for Simulation-Based Computer Architecture Research," *IEEE Computer Architecture Letters*, vol. 1, no. 1, Jan.–Dec. 2002.

[Kuon 07] I. Kuon and J. Rose, "Measuring the Gap Between FPGAs and ASICs," *IEEE Trans. Comput.-Aided Des. Integr. Circuits and Syst.,* vol. 26, no. 2, pp. 203–215, Feb. 2007.

[Leon] Aeroflex Gaisler, "Leon3 Processor," http://www.gaisler.com.

[Li 09] M.-L. Li *et al.*, "Accurate Microarchitecture-level Fault Modeling for Studying Hardware Faults," *Proc. IEEE Intl. Symp. High Performance Computer Architecture*, pp. 105–116, 2009.

[Li 13] Y. Li, E. Cheng, S. Makar, and S. Mitra, "Self-Repair of Uncore Components in Robust System-on-Chips: An OpenSPARC T2 Case Study," *Proc. IEEE Intl. Test Conf.,* 2013.

[Lilja 13] K. Lilja *et al.*, "Single-Event Performance and Layout Optimization of Flip-Flops in a 28-nm Bulk Technology," *IEEE Trans. Nucli. Sci.*, vol. 60, no. 4, pp. 2782–2788, Aug. 2013.

[Lin 06] I.-C. Lin, S. Srinivasan, and N. Vijaykrishnan, "Transaction Level Error Susceptibility Model for Bus Based SoC Architectures," *Proc. Intl. Symp. Quality Electronic Design*, pp. 775–780, 2006.

[Lin 14] D. Lin *et al.*, "Effective Post-Silicon Validation of System-on-Chips Using Quick Error Detection," *IEEE Trans. Comput.-Aided Des. Integr. Circuits and Syst.,* vol. 33, no. 10, pp. 1573–1590, Oct. 2014.

[Lin 15] D. Lin *et al.*, "Quick Error Detection Tests with Fast Runtimes for Effective Post-Silicon Validation and Debug," *Proc. Design, Automation and Test in Europe*, pp. 1168–1173, 2015.

[Loveless 11] T. D. Loveless *et al.*, "Neutron- and Proton-Induced Single Event Upsets for D- and DICE-Flip/Flop Designs at a 40 nm Technology Node," *IEEE Trans. Nucli. Sci.*, vol. 58, no. 3, pp. 1008–1014, June 2011.

[Maniatakos 11a] M. Maniatakos, C. Tirumurti, A. Jas, and Y. Makris, "AVF Analysis Acceleration via Hierarchical Fault Pruning," *Proc. European Test Symposium*, pp. 87–92, 2011.

[Maniatakos 11b] M. Maniatakos *et al.*, "Instruction-Level Impact Analysis of Low-Level Faults in a Modern Microprocessor Controller," *IEEE Trans. Computers*, vol. 60, no. 9, pp. 1260–1273, Sept. 2011.

[McCluskey 71] E. J. McCluskey and F. W. Clegg, "Fault Equivalence in Combinational Logic Networks," *IEEE Trans. Computers*, vol. 20, no. 11, pp. 1286–1293, Nov. 1971.

[McCluskey 00] E. J. McCluskey and C.-W. Tseng, "Stuck-Fault Tests vs. Actual Defects," *IEEE Intl. Test Conf.*, pp. 336–343, 2000.

[Meaney 05] P. J. Meaney, S. B. Swaney, P. N. Sanda, and L. Spainhower, "IBM z990 soft error detection and recovery," *IEEE Trans. Device and Materials Reliability*, vol. 5, no. 3, pp. 419–427, Sept. 2005.

[Meixner 07] A. Meixner, M. E. Bauer, and D. J. Sorin, "Argus: Low-Cost, Comprehensive Error Detection in Simple Cores," *Proc. Intl. Symp. Microarchitecture*, pp. 210–222, 2007.

[Michalak 12] S. E. Michalak *et al.*, "Assessment of the Impact of Cosmic-Ray-Induced Neutrons on Hardware in the Roadrunner Supercomputer," *IEEE Trans. Device and Materials Reliability*, vol. 12, no. 2, pp. 445–454, June 2012.

[Mirkhani 14] S. Mirkhani, H. Cho, S. Mitra, and J. A. Abraham, "Rethinking Error Injection for Effective Resilience," *Proc. IEEE Asia and South Pacific Design Automation Conf.*, pp. 390–393, 2014.

[Miskov-Zivanov 10] N. Miskov-Zivanov and D. Marculescu, "Multiple Transient Faults in Combinational and Sequential Circuits: A Systematic Approach," *IEEE Trans. Comput.-Aided Des. Integr. Circuits and Syst.*, vol. 29, no. 10, pp. 1614–1627, Oct. 2010.

[Mitra 00] S. Mitra and E. J. McCluskey, "Which Concurrent Error Detection Scheme to Choose?," *Proc. Intl. Test Conf.*, pp. 985–994, 2000.

[Mitra 05] S. Mitra *et al.*, "Robust System Design with Built-In Soft-Error Resilience," *IEEE Computer*, vol. 38, no. 2, pp. 43–52, Feb. 2005.

[Mitra 10] S. Mitra, K. Brelsford, and P. N. Sanda, "Cross-Layer Resilience Challenges: Metrics and Optimization," *Proc. Design, Automation and Test in Europe*, pp. 1029–1034, 2010.

[Mitra 14] S. Mitra *et al.*, "The Resilience Wall: Cross-Layer Solution Strategies," *Proc. IEEE Intl. Symp. VLSI Design, Automation and Test*, 2014.

[Mukherjee 05] S. S. Mukherjee, J. Emer, and S. K. Reinhardt, "The Soft Error Problem: An Architectural Perspective," *Proc. IEEE Intl. Symp. High Performance Computer Architecture*, pp. 243–247, 2005.

[Mukherjee 08] S. S. Mukherjee, "Architecture Design for Soft Errors," Morgan Kaufmann, 2008.

[Nakano 06] J. Nakano, P. Montesinos, K. Gharachorloo, and J. Torrellas, "ReViveI/O: Efficient Handling of I/O in Highly-Available Rollback-Recovery Servers," *Proc. Intl. Symp. High-Performance Computer Architecture*, pp. 200–211, 2006.

[Oh 02] N. Oh, P. P. Shirvani, and E. J. McCluskey, "Error Detection by Duplicated Instructions in Super-Scalar Processors," *IEEE. Trans. Reliability*, vol. 51, no. 1, pp. 63–75, Mar. 2002.

[Oldiges 09] P. Oldiges *et al.*, "Technologies to further reduce soft error susceptibility in SOI," *Proc. IEEE Intl. Electron Devices Meeting*, 2009.

[OpenSPARC] "OpenSPARC: World's First Free 64-bit Microprocessor," http://www.opensparc.net.

[Pattabiraman 11] K. Pattabiraman *et al.*, "Automated Derivation of Application-Specific Error Detectors Using Dynamic Analysis," *IEEE Trans. Dependable and Secure Computing*, vol. 8, no. 5, pp. 640–655, Sept.–Oct. 2011.

[Pellegrini 12] A. Pellegrini *et al.*, "CrashTest'ing SWAT: Accurate, Gate-Level Evaluation of Symptom-Based Resiliency Solutions," *Proc. Design, Automation and Test in Europe*, pp. 1106–1109, 2012.

[Prvulovic 02] M. Prvulovic, Z. Zhang, and J. Torrellas, "ReVive: Cost-Effective Architectural Support for Rollback Recovery in Shared-Memory Multiprocessors," *Proc. Intl. Symp. Computer Architecture*, pp. 111–122, 2002.

[Quinn 13] H. M. Quinn, D. A. Black, W. H. Robinson, and S. P. Buchner, "Fault Simulation and Emulation Tools to Augment Radiation-Hardness Assurance Testing," *IEEE Trans. Nucli. Sci.,* vol. 60, no. 3, pp. 2119–2142, June 2013.

[Racunas 07] P. Racunas, K. Constantinides, S. Manne, and S. S. Mukherjee, "Perturbation-based Fault Screening," *Proc. IEEE Intl. Symp. High Performance Computer Architecture,* pp. 169–180, 2007.

[Ramachandran 08] P. Ramachandran *et al.,* "Statistical Fault Injection," *Proc. IEEE Intl. Conf. Dependable Systems and Networks,* pp. 122–127, 2008.

[Rebaudengo 02] M. Rebaudengo, M. S. Reorda, and M. Violante, "Analysis of SEU effects in a pipelined processor," *Proc. IEEE Intl. On-Line Testing Workshop,* pp. 112–116, 2002.

[Rimen 94] M. Rimen, J. Ohlsson, and J. Torin, "On microprocessor error behavior modeling," *Proc. IEEE Intl. Symp. Fault-Tolerant Computing,* pp. 76–85, 1994.

[Sanda 08] P. N. Sanda *et al.*, "Soft-error resilience of the IBM POWER6 processor," *IBM Journal of Research and Development,* vol. 52, no. 3, pp. 275–284, May 2008.

[Schelle 10] G. Schelle *et al.*, "Intel Nehalem Processor Core Made FPGA Synthesizable," *Proc. Intl. Symp. Field Programmable Gate Arrays,* pp. 3–12, 2010.

[Seifert 10] N. Seifert, "Radiation-induced Soft Errors: A Chip-level Modeling Perspective," *Foundat. Trends® in Electron. Design Autom.,* vol. 4, no. 2–3, pp. 99–221, Feb. 2010.

[Seifert 12] N. Seifert *et al.,* "Soft Error Susceptibilities of 22 nm Tri-Gate Devices," *IEEE Trans. Nucli. Sci.,* vol. 59, no. 6, pp. 2666–2673, Dec. 2012.

[Simics] "Windriver Simics Full System Simulation," http://www.windriver.com/products/simics/.

[Smolens 04] J. C. Smolens *et al.*, "Fingerprinting: Bounding Soft-error Detection Latency and Bandwidth," *Proc. Intl. Conf. Architectural Support for Programming Languages and Operating Systems,* pp. 224–234, 2004.

[Sorin 02] D. J. Sorin, M. M. K. Martin, M. D. Hill, and D. A. Wood, "SafetyNet: Improving the Availability of Shared Memory Multiprocessors with Global Checkpoint/Recovery," *Proc. Intl. Symp. Computer Architecture,* pp. 123–134, 2002.

[Wang 04] N. J. Wang, J. Quek, T. M. Rafacz, and S. J. Patel, "Characterizing the Effects of Transient Faults on a High-Performance Processor Pipeline," *Proc. Intl. Conf. on Dependable Systems and Networks*, pp. 61–70, 2004.

[Wang 07] N. J. Wang, A. Mahesri, and S. J. Patel, "Examining ACE Analysis Reliability Estimates Using Fault-Injection," *Proc. Intl. Symp. Computer Architecture*, pp. 460–469, 2007.

[Weaver 08] D. L. Weaver, "OpenSPARC internals: OpenSPARC T1/T2 CMT throughput computing," Sun Microsystems, 2008.

[Woo 95] S. C. Woo *et al.*, "The SPLASH-2 Programs: Characterization and Methodological Considerations," *Proc. Intl. Symp. Computer Architecture*, pp. 24–36, 1995.

[Yim 10] K. S. Yim, Z. Kalbarczyk, and R. K. Iyer, "Measurement-based Analysis of Fault and Error Sensitivities of Dynamic Memory," *Proc. IEEE/IFIP Intl. Conf. on Dependable Systems and Networks*, pp. 431–436, 2010.

[Yoo 09] R. M. Yoo, A. Romano, and C. Kozyrakis, “Phoenix rebirth: Scalable MapReduce on a large-scale shared-memory system,” *Proc. Intl. Symp. Workload Characterization*, pp. 198–207, 2009.

[Zhang 10] Y. Zhang, J. W. Lee, N. P. Johnson, and D. I. August, “DAFT: Decoupled Acyclic Fault Tolerance,” *Proc. Intl. Conf. Parallel Architectures and Compilation Techniques*, pp. 87–98, 2010.