NORTHEASTERN UNIVERSITY
Graduate School of Engineering

**Thesis Title:** High Level Modeling and Mitigation of Transient Errors in Nano-scale Systems

**Author:** Syed Zafar Shazli

**Department:** Electrical and Computer Engineering

Approved for Thesis Requirements of the Doctor of Philosophy Degree:

Thesis Advisor: Prof. Mehdi Tahoori  
Date

Thesis Committee: Prof. David Kaeli  
Date

Thesis Committee: Prof. Ningfang Mi  
Date

Department Chair: Prof. Ali Abur  
Date

Graduate School Notified of Acceptance:

Director of the Graduate School
Copy Deposited in Library:

Reference Librarian  
Date
HIGH LEVEL MODELING AND MITIGATION OF TRANSIENT ERRORS IN NANO-SCALE SYSTEMS

A Thesis Presented
by

Syed Zafar Shazli

to
The Department of Electrical and Computer Engineering
in partial fulfillment of the requirements
for the degree of

Doctor of Philosophy
in
Electrical Engineering
in the field of
Computer Engineering

Northeastern University
Boston, Massachusetts

January 2011
© Copyright 2011 by Syed Zafar Shazli
All Rights Reserved
Abstract

Soft errors, due to cosmic radiations, are a major reliability barrier for VLSI designs. The vulnerability of such systems to soft errors grows exponentially with technology scaling. To meet reliability constraints in a cost-effective way, it is critical to assess soft error reliability parameters in early design stages in order to optimize reliability in the entire design cycle. Unlike soft error modeling for gate-level netlists, soft error propagation models for high-level behavioral designs are not straightforward. We divide the work done into three parts. First, the Soft Error Rate (SER) computation problem is modeled as a Boolean Satisfiability (SAT) problem and SAT solvers are used to compute SER for combinational and sequential circuits. SAT is also used to compute a metric called Hardware Vulnerability Factor (HVF). HVF is the probability that an error in any bit of the internal processor structure will result in an error in a program visible state. The HVF computation problem is transformed into an equivalent Boolean satisfiability problem and state-of-the-art SAT solvers are used to obtain HVF for a 5-stage MIPS pipeline. Next, several schemes are proposed for detecting, correcting and recovering from soft errors in processor pipelines. Two types of pipelines are considered. One is a simple 5-stage MIPS pipeline, while the other is a superscalar pipeline similar to the ALPHA processor. Lastly, a case study involving thousands of high-availability systems is presented. The study considers, soft errors occurring in the processors used in these systems.
Acknowledgements

After thanking the Almighty for everything He gave, I would like to thank everyone who helped and inspired me in my doctoral studies.

In particular, I am heartily thankful to my thesis supervisor Prof Mehdi Tahoori, whose guidance, encouragement and support throughout my research enabled me to develop an understanding and appreciation of the subject. He was always accessible and willing to help in all academic matters.

I was honoured to interact with Prof David Kaeli whose insights in Computer Architecture made much of this work possible. I consider him an icon of a world-class researcher for his efforts and passion on research.

Thanks are also due to Prof Ningfang Mi for serving on the Thesis committee and giving useful advice.

I acknowledge the support of Graduate School of Engineering, Northeastern University, for providing excellent resources that helped me in carrying out this work.

All the members of the Dependable Nanocomputing Lab, Cihan, Masoud, Navid and Liang provided much needed support and stimulating ideas. I am also grateful for the help and encouragement provided by Kevin Granlund and Hossein Asadi at EMC Corporation while I worked there on internships and coop. Kevin was my Manager and mentor and helped me grasp many challenging issues in reliability and performance in high-availability systems.

My deepest gratitude goes to my parents for their love and support throughout my
life. My wife Ayesha, and kids, Hannnah and Abdurrahman, supported me throughout my doctoral studies and endured all the challenges that come with Graduate education. I considered them my support staff, and they were always there for encouragement and support. I am pretty sure that I would never have been able to carry out this work without their constant love and determination.
Contents

Abstract iv

Acknowledgements v

1 Introduction 1
  1.1 Scope and Contributions 3
  1.2 Outline 5

2 Soft Errors: Origins, Terminology and Recent Trends 6
  2.1 Origins and Physics of Soft Errors 6
  2.2 Effects of Soft Errors on present day systems 9
  2.3 Technology Trends 10
  2.4 Terminology 12

3 SAT based schemes for SER computation 14
  3.1 Background 15
  3.2 Related Work 16
  3.3 Methodology for Combinational and Sequential Circuits 18
  3.4 SAT-based Computation of Hardware Vulnerability Factor 36
  3.5 Summary 44
# 4 Soft Errors in Processor Pipelines

4.1 Background .................................................. 47
4.2 Related Work ............................................... 50
4.3 Soft Error Detection and Recovery in Inorder Pipelines .......... 57
4.4 Error Detection and Recovery in Superscalar Pipelines ........... 65
4.5 Summary .................................................... 85

# 5 Soft Error Field Failure Analysis

5.1 Related Work ............................................... 87
5.2 Information Systems ....................................... 89
5.3 Methodology ................................................ 90
5.4 Examples ................................................... 95
5.5 Analysis of SEU Events .................................... 98
5.6 Comparison of SEU and non-SEU events ........................ 101
5.7 Analysis of Results ........................................ 108
5.8 Summary .................................................... 111

# 6 Conclusions

# Bibliography
List of Figures

2.1 A charged particle causes ionization and soft error ........................................ 7
2.2 Relative SER vs. process nodes ................................................................. 10
2.3 Soft Errors in SRAM, latch and Logic .................................................. 11

3.1 Algorithm for the SAT-based SER Computation Methodology ................. 20
3.2 Fault injection in a sample circuit ............................................................. 20
3.3 Fault injection in a 4-bit subtracter .......................................................... 22
3.4 Unrolling a sequential circuit ................................................................. 23
3.5 Procedure for computing SER in Sequential Circuits .............................. 25
3.6 The implementation flow chart of SAT-based methodology .................... 28
3.7 The implementation flow chart of the fault simulation methodology ........ 29
3.8 SAT-based approach for HVF Computation ........................................... 39
3.9 Flow from Behavioral Verilog to CNF ....................................................... 39
3.10 HVF distribution for different resources .................................................. 41
3.11 Distribution of bits with highest HVF ..................................................... 41
3.12 Distribution of flip-flops (bits) based on HVF ........................................ 42
3.13 HVF of different instruction classes ....................................................... 43
3.14 HVF distribution for UCore processor ...................................................... 43
3.15 Distribution of bits with highest HVF ..................................................... 44
3.16 Distribution of flip-flops (bits) based on HVF for the UCore processor .... 44
4.1 The MIPS 5-stage pipeline .............................................. 48
4.2 The microarchitecture of a superscalar pipeline .................. 51
4.3 Proposed error recovery architecture (one stage) .................. 60
4.4 Block diagram of the centralized ECC unit ......................... 61
4.5 An example of 5-stage MIPS pipeline with data forwarding from ALU .... 62
4.6 An Example of Error Recovery ....................................... 63
4.7 Lifetime of an instruction in IQ, ROB and Rename table ........... 67
4.8 Error detection at commit and issue times ............................ 68
4.9 Updating the regHT register ........................................... 70
4.10 Logic for Error Detection in Head and Tail pointers ......... 71
4.11 Unified Error Correction Logic ....................................... 74
4.12 Effect on Peak and Average Power for SPEC2000 Benchmarks ... 83
4.13 Effect of changing IQ size on Performance .......................... 84
4.14 Effect of changing IQ size on Average Power ..................... 84

5.1 A flowchart of the methodology ....................................... 91
5.2 Classification of SEU events ........................................... 92
5.3 Distribution of SEUs in the two systems .............................. 98
5.4 Distribution of various causes of RIEs in different systems ...... 102
5.5 Severity of RIEs due to various causes in different systems ... 104
5.6 Probability and Reliability plots for Hardware failures .......... 106
5.7 Probability and Reliability plots for Software failures .......... 108
5.8 Probability and Reliability plots for Power failures ............. 109
5.9 Weibull (a) and Normal (b) Probability plots for Transient Failures. 109
5.10 Reliability against time for Transient Failures .................. 110
List of Tables

3.1 Comparison of SAT-flow and fault simulation for combinational circuits . . 30
3.2 Validation results using small-sized sequential circuits . . . . . . . . . . . . 32
3.3 Comparison of SAT-flow and random fault simulation for sequential circuits 34
3.4 Experiments for Large sequential circuits . . . . . . . . . . . . . . . . . . . 35

4.1 Area and timing of the Original and Modified MIPS . . . . . . . . . . . . . 65
4.2 The configuration of our superscalar implementation . . . . . . . . . . . . . 77
4.3 Performance overhead for SPEC2000 benchmarks . . . . . . . . . . . . . . 81
4.4 Area overhead of error detection and correction logic . . . . . . . . . . . . . 82

5.1 Statistics for Machine Checks in Systems A . . . . . . . . . . . . . . . . . . 100
5.2 Relative FIT rates for Systems A and B . . . . . . . . . . . . . . . . . . . . 100
5.3 Characterizing the causes of RIEs . . . . . . . . . . . . . . . . . . . . . . . . 101
5.4 Distribution of systems in the sample . . . . . . . . . . . . . . . . . . . . . . 105
5.5 $R^2$ values for Hardware Failures . . . . . . . . . . . . . . . . . . . . . . . . 105
5.6 $R^2$ values for Software Failures . . . . . . . . . . . . . . . . . . . . . . . . 107
5.7 $R^2$ values for Power Failures . . . . . . . . . . . . . . . . . . . . . . . . . . 107
5.8 $R^2$ values for Transient Failures . . . . . . . . . . . . . . . . . . . . . . . . 110
Chapter 1

Introduction

The advancement of semiconductor technologies has enabled electronic circuits and systems to penetrate every area of contemporary life, from commodity devices like cellular phones to critical high availability systems such as those used in aircrafts and banks. While malfunctions in the former may cause no harm other than inconvenience, the slightest malfunction in the latter may have catastrophic consequences. These malfunctions can occur either due to permanent defects (like wear and tear and manufacturing anomalies) or due to transient errors (like electromagnetic interference, power glitches and cosmic radiations). A soft error occurs when a radiation event causes enough of a charge disturbance to reverse or flip the data state of a logic gate, memory cell, register, latch, or flip-flop.

Soft errors, also called single event upsets (SEUs) are radiation-induced errors caused by neutrons from cosmic rays and alpha particles from packaging material. In the past, soft errors were regarded as a major concern only for space applications. Presently, for designs manufactured at advanced technology nodes, system-level soft errors are much more frequent than in the previous generations. The vulnerability of VLSI systems to soft errors exponentially increases as an unwanted side effect of Moore’s law. The error is soft because the circuit/device itself is not permanently damaged by the radiation.

The International Technology Roadmap for Semiconductors (ITRS) predicts that there
CHAPTER 1. INTRODUCTION

is an exponential growth in integration levels in computer systems \cite{2}. According to the roadmap, the device sizes will be around 32nm by 2013. As feature sizes shrink, the amount of charge per device decreases, and so a particle strike is much more likely to cause an error. Particles of lower energy, which are more abundant than high energy particles, will generate sufficient charge to cause a soft error. In the absence of error correction schemes, the system error rate will grow in direct proportion to the number of bits on the chip. Soft errors are emerging as a significant obstacle to increasing microprocessor transistor count in future process technologies. Although soft error rates of individual transistors are not projected to rise, incorporating more transistors into the same device real estate makes a device more likely to encounter a fault. As a result, it is expected that maintaining microprocessor error rates at acceptable levels will require specific design changes \cite{3}.

In the last decade or so, mainstream server vendors have experienced several hardware failures in the field that have been attributed to transient faults \cite{4} \cite{5} \cite{6} \cite{7}. Leading technology experts began to warn designers that device reliability will wane in the 45nm regime and beyond \cite{8} \cite{9}. Moreover, increased stress on availability of high performance systems has resulted in an expansion in fault tolerance research to include the effects of transient faults. It must be noted here, that these faults have different implications for system design than permanent faults. For instance, a transient fault does not indicate an underlying hardware component failure; once the fault is cleared (using reboot for example), the hardware will resume normal functioning. Although SEUs do not break the silicon, their effect is a logic glitch that can potentially corrupt combinational logic computation or state bits. Although a variety of studies have shown that demonstrate the unlikeliness of such events at ground level, the researchers in architecture and circuit communities are concerned because of continued reduction in feature sizes and supply voltage, both of which exacerbate a design’s vulnerability to SEUs.

In the past, architects have often left reliability concerns to lower levels of system stack. As the severity of these problems increases, however, low-level solutions will likely not suffice
and straightforward system-level solutions such as blind redundancy will likely become too expensive [10]. Hence, there is a need to make reliability an initial design constraint; along with other design constraints such as cost, performance, and power consumption; and explore cost-effective approaches to reliability aware designs.

1.1 Scope and Contributions

In this thesis, we handle the soft error problem at behavioral and system levels. We have demonstrated the use of Boolean Satisfiability (SAT) approaches, to model soft errors in early design stages. Our proposed methods provide accurate soft error rates of combinational and sequential circuits described at the behavioral level. The use of SAT has also been extended to obtain Hardware Vulnerability Factor (HVF) for various stages of an inorder processor pipeline. HVF is the probability that an error in any bit of the internal processor structure will result in an error in a program visible state. For more complex, superscalar pipelines, we have proposed various error detection and recovery techniques to protect various front-end structures. Additionally, we analyze the results of a case study involving several thousand high-availability information systems present in the field. The study looks at the effect of soft errors on these systems and compares their relative occurrence with non-SEU errors.

In particular, the main contributions of this work are as follows:

- A framework is presented to accurately obtain soft error rate (SER) for high-level (behavioral) descriptions (Verilog or VHDL) in early design stages. The SER problem is transformed into an equivalent Boolean satisfiability problem and state-of-the-art SAT-solvers have been used to obtain SER. Experimental results on benchmark circuits show the applicability of the approach to medium-sized and large-sized circuits.

- An automated flow has been developed to convert combinational and sequential behavioral descriptions into equivalent SAT instances. The flow has been extensively
validated by exhaustive fault simulation of several benchmark circuits. It can be used to obtain accurate soft error rates of circuits described at behavioral level.

- Hardware Vulnerability Factor of various structures in an inorder processor pipeline has been obtained using SAT-based techniques. The structures include different units like Fetch, Decode, Execute etc. The processor description is at the behavioral (Verilog) level and experiments have been conducted on two processor cores.

- Per-instruction HVF is obtained for various types of MIPS instructions. 54 MIPS instructions are grouped into arithmetic, logical, branch/jump, and store/load. The experimental results suggest that HVF variation for different instructions is almost negligible.

- A methodology has been designed and implemented for transient error detection and recovery in in-order processor pipelines. The technique uses Error Correcting Codes (ECC) for error correction. The synthesis results show an area overhead of around 15%.

- Several error detection, correction and recovery techniques have been designed and implemented, for protection against runtime errors in front-end structures of a superscalar processor pipeline. The implementation is done on top of an RT-Level model of Alpha processor as well as on a cycle-accurate performance simulator. The techniques are able to detect and recover from all single bit upsets occurring in the front-end structures, with only 8% increase in Cycles per Instruction (CPI), 3.4% in critical path delay, 0.5% area overhead for the entire core, and less than 2% power overhead.

- A field analysis has been carried out on thousands of live high-availability systems. The analysis looks at the relative occurrence and severity of soft error related incidents compared to other permanent and temporary errors. The focus is on the SEUs.
occurring in state-of-the-art microprocessors used in high performance information systems. The results agree with previously developed analytical models, and show that the size of on-chip cache memory has a direct impact on the system-level failure rate and

1.2 Outline

This thesis summarizes my PhD study for modeling transient errors at higher levels of abstraction. The remainder of this thesis is organized as follows:

Chapter 2 contains the terminology and related work done in the area of soft error modeling and mitigation. We look at previous approaches used in the literature for SER estimation, fault tolerance in processors and field failure analysis of information systems.

Chapter 3 looks at SAT-based methods to compute SER of combinational and sequential circuits described at behavioral level. It also includes techniques to obtain HVF of various structures in the processor pipeline. Implementation details and experimental results are discussed as well.

In Chapter 4, we look at soft error detection, correction and recovery techniques in in-order cores as well as superscalar out-of-order pipelines. Algorithms for error correction and recovery of single bit errors in various fields of front-end structures are presented. The chapter also discusses the area, performance and power overheads of the proposed schemes.

A case study considering SEUs occurring in high-availability information systems is presented in Chapter 5. Detailed steps of the methodology undertaken to carry out the field study are provided along with some examples from logs of live systems. Probability plots are also shown, in order to provide a graphical picture of how well the data fits well-known failure distributions.

Finally, Chapter 6 concludes the thesis. It also contains directions of future research.
Chapter 2

Soft Errors: Origins, Terminology and Recent Trends

As silicon technologies move into the nanometer regime, there is a growing concern for the reliability of transistor devices. Leading technology experts have begun to warn designers that device reliability will wane in the 45nm regime and beyond [11]. In fact, device scaling aggravates a number of long standing silicon failure mechanisms, and it introduces a number of new non-trivial failure modes. Unless these reliability concerns are addressed, either through on-line detection and correction or introduction of more robust devices, component yield and lifetime will soon be compromised.

2.1 Origins and Physics of Soft Errors

When an energetic particle (alpha particles from packaging or neutrons from cosmic rays), strikes a CMOS transistor, it induces a localized ionization capable to reverse (flip) the data state of a memory cell, logic gate, latch, or flipflop. This is called a Single Event Upset (SEU) or Soft Error [4]. The errors are called soft since the circuit itself is not permanently damaged by the radiation. If the system is reset and rerun, the hardware will perform
Figure 2.1: A charged particle causes ionization and soft error correctly. The frequency of occurrence of the errors induced by SEU is commonly referred to as soft error rate (SER).

Figure 2.1 illustrates the phenomenon of soft errors. A high energy particle enters the substrate near the drain of a transistor. This particle interacts with the substrate and causes many electron - hole pairs to be formed. The holes are quickly swept away to the bulk node, however the electrons are collected by the drain node, which we assume to be at a high voltage. These electrons will cause the voltage at the drain node to drop. The magnitude of the voltage drop depends on the charge collected. If the amount of charge collected \( Q_{\text{coll}} \) exceeds an amount known as the critical charge \( Q_{\text{crit}} \) an error will occur \[12\].

In the past three decades, researchers have discovered several mechanisms that cause soft errors in semiconductor devices especially at terrestrial altitudes.

**Alpha Particles**

May and Woods reported on alpha particle - induced soft errors in 16MB DRAMs in the late 1970s \[13\]. This was the first public account of radiation - induced upsets in electronic devices at sea level. They determined that these errors were caused by \( \alpha \) particles, emitted in the radioactive decay of uranium and thorium impurities in packaging materials. These impurities are present just in few parts per million levels and emit alpha particles at specific discrete energies over a range from 4 to 9 MeV. When an alpha particle travels through a semiconductor device, made of semiconductor materials, it loses its kinetic energy predominantly through interactions with the material electrons and thus leaves a trail of
ionization in the field. The higher the energy of the alpha particle, the farther it travels before being stopped by the material. The distance required to stop an alpha particle is a function of both its energy and the properties of the material in which it is traveling. In silicon, the range for a 10-MeV alpha particle is less than 100 µm. Thus, alpha particles from outside the packaged device are clearly not a concern; only alpha particles emitted by the device materials (semiconductor materials) and packaging materials need to be considered. In the majority of CMOS devices, if semiconductor manufacturing materials and packaging materials could be purified sufficiently, the fraction of soft errors due to alpha particles would fall to very low levels [14].

High Energy Neutrons

In the mid-1990s, due to the usage of packaging materials with low alpha emission rates, high-energy cosmic radiations proved to be the dominant source of soft errors in DRAM devices [5]. These particles can induce soft errors in semiconductor devices via the secondary ions produced by the neutron reactions with the silicon nuclei. Neutrons are one of the higher flux components, and their reaction with devices is more likely to cause upsets in the circuits at terrestrial altitudes. The neutron flux is strongly dependent on the altitude since the intensity of the cosmic ray neutron flux increases with altitude. For example, going from the sea level to 10,000 feet above, the cosmic ray flux increases 10 times (this trend starts to saturate at about 50,000 feet) [14]. Due to proton shielding effects induced by interactions with the Earth’s magnetic field, the neutron flux is also dependent on the magnetic rigidity and the geographical location. However, neutrons do not directly generate ionization in the silicon and their interaction with chip materials is purely kinetic. Neutrons tend to generate more charge than alpha particles. Typical waveforms are tens of femto-coulombs within a short period of a few to tens of picoseconds. Unlike alpha particles, the cosmic neutron flux cannot be reduced significantly with shielding or high purity materials. Concrete has been shown to shield the cosmic radiations. Two feet of concrete thickness around the device would reduce neutron flux by 50%. However, this approach (shielding by concrete) is not
feasible for portable devices and design changes or process modification are necessary.

**Other Sources**

Radiations induced from the interaction of low-energy cosmic ray neutrons and boron can be another source of soft errors, especially in smaller technology nodes. Low-energy neutrons have energy of less than 1 MeV. Some other sources such as electromagnetic interference (EMI), ground-bounce, electromigration, hot carrier injection, cross-talk, negative biased temperature instability (NBTI) and process variation also cause transient errors in the systems [15].

### 2.2 Effects of Soft Errors on present day systems

It is important to note, that even if the background radiation does not exceed usual levels, soft errors can cause serious problems in digital systems. In the last decade, this has been demonstrated by at least 2 major issues related to radiation induced errors.

The first case is the problem in the high end server line 'Enterprise' of Sun Systems in 2000 as reported in [16]. The Enterprise was a high-end server in Sun’s product line with cost price ranging from $50,000 to $1 Million. During 1999, some customers reported that occasionally, the server crashed for no apparent reason. This was a serious problem for web-based companies which were supposed to be online 24/7. One customer reported that their server had crashed and rebooted 4 times in a few months.

After months of tedious investigations, it was finally identified that soft errors in cache memory of the server were the root cause. The cache modules contained SRAM chips vulnerable to soft errors. With aggressive technology scaling and increase in number of bits per SRAM chip, the soft error rate of the complete system increased from one product generation to the next, until the point where soft errors became the dominant source of system failures.

Sun stated that the issue had cost tens of millions of dollars and a huge amount of
Figure 2.2: Relative SER vs. process nodes.

man-hours. The case demonstrated that in the absence of adequate mechanisms, soft errors could cause serious problems, especially for systems that contain large amount of memories.

Another case of a major soft error issue was with the routers of Cisco systems in 2003 \[17\]. Router cards of the 12000 series, with a price tag of about $200,000, showed failures caused by radiation induced soft errors. Parity errors were observed in both memory and ASICs. The systems however, recovered using reboots.

### 2.3 Technology Trends

An important trend in present designs is variability due to smaller feature sizes. Although variability is generally considered to be responsible for leakage etc., it also significantly reduces the reliability of devices. At larger geometries, such as the 250nm node, the probability of soft errors (i.e. transistors switching incorrectly) is relatively inconsequential. As can be seen in Figure 2.2, this is not the case for 32nm and beyond.

It is expected that in the future, both transistor size and critical charge will continue to decrease with scaling of the manufacturing processes. The number of particles able to induce errors, for a given die area, is expected to saturate, SER being determined by the actual number of strikes. After the saturation phase the SER/bit tends to decrease.
Figure 2.3: Soft Errors in SRAM, latch and Logic \[\text{1}\]

However, the number of devices will continue to increase, due to the higher complexity and improved functionality of the integrated circuits. This evolution will lead to higher SER/chip. Likelihood of particle induced multi-bit errors will also increase, as well as probability of soft errors within combinational logic \[\text{11}\].

The soft error robustness of logic has begun to emerge as a significant concern. As technology scales the critical charge of SRAM cells has decreased, however the critical charge of logic circuits has decreased at a greater rate. In past technologies where the critical charge of logic used to be an order of magnitude greater than SRAM cells, now they are almost identical, and going forward logic might well be more vulnerable than SRAM.

Figure\[\text{2.3}\] shows that the $Q_{\text{crit}}$ of logic circuits decreases more rapidly with feature size than the $Q_{\text{crit}}$ of memory elements. Since the y-axis of this graph is log scale, the actual decline is exponentially greater across this range of feature sizes. This steep reduction in $Q_{\text{crit}}$ is primarily due to quadratic decrease in node capacitance with feature size. Logic transistors are typically wider than transistors used in memory circuits, where density is important, and thus this effect is more pronounced in logic circuits.

Although, protecting cache and memory against transient errors has received greater
attention in recent years [18] [19], not much effort has been spent on protecting other parts of the processor core from these errors. Trends in processor architecture have shown an increasing use of Instruction Level Parallelism (ILP) to improve performance, including superscalar and multi-core processors [20]. Most modern processors like [21] [22] employ deep pipelines and try to issue multiple instructions per cycle. Hence, its increasingly important that error detection and recovery mechanisms be explored to protect the pipelines from transient effects.

2.4 Terminology

We begin by making a distinction between a fault, an error, and a failure. We define a fault as the result of a raw event such as a single-event upset. An error is one possible result of a fault, and is an event that causes a decrease in a system’s fault tolerance. Finally, a system failure is an event that causes the system to incorrectly process a task, or to stop responding to requests, and is one possible result of an error. As we will see, not all faults lead to an error; and not all errors lead to failures. For example, a fault might lead to an error in a system’s register file, regardless of whether it actually causes a system failure. If the register file was part of a lockstepped core-pair, for example, the system would detect and recover from this error; however, the system might not be able to tolerate a fault in the second register file, resulting in decreased fault tolerance during that time period.

Errors are often classified as detected or undetected. An undetected error might result in a Silent Data Corruption (SDC) failure. An SDC is a corruption of system state that is unreported to either the system or the program. This is generally regarded as the most severe failure that can result from an error. A detected error can be further classified as a Corrected Error (CE) or a Detected Uncorrected Error (DUE) [3]. Corrected Errors are errors from which recovery to normal system operation is possible, either by hardware or software. Detected Uncorrected Errors are errors that are discovered and reported, but
from which recovery is not possible. These errors typically cause a program or system to crash.

The raw fault rate of a system to a particular class of fault is the number of faults of that type per unit time. This is typically expressed in units of Failures-In-Time (FIT); one FIT is equal to one failure in a billion hours. The error rate of a system is defined as the number of errors per unit time, also expressed in FIT.

Finally, we draw a distinction between two metrics to measure overall fault tolerance: reliability and availability. A system’s reliability can be defined as the fraction of initiated jobs that complete correctly. A system’s availability is the fraction of (wall-clock) time that a system is able to initiate jobs. Both metrics are a function of the system’s error rate and its error handling infrastructure. The relative importance of these metrics differs based on the usage model of the system; for some systems (e.g., servers with many small tasks), maintaining high availability can be more important than correctly completing a particular individual job.
Chapter 3

SAT based schemes for SER computation

In modern high-performance systems, reliability constraints are becoming more and more stringent. Fast, accurate, and efficient estimation of Soft Error rates is necessary as a reliability metric during the entire design cycle. As mentioned in [23], not all particle strikes result in an observable failure in the system. The effect of injected charge can be masked at multiple levels such as electrical masking, timing-window masking, logical masking, and architectural masking.

Logical masking occurs when a particle strikes a portion of the combinational logic that is blocked from affecting the output due to a subsequent gate whose result is completely determined by its other input values. Electrical masking occurs when the pulse resulting from a particle strike is attenuated by subsequent logic gates due to the electrical properties of the gates to the point that it does not affect the result of the circuit. Timing-window masking occurs when the pulse resulting from a particle strike reaches a latch, but not at the clock transition where the latch captures its input value. These effects could however, reduce significantly with decreasing feature sizes and increasing number of stages in the processor pipeline [11].
Traditionally, SER estimation has been done on transistor level, gate level and architectural level models. Transistor level SER modeling methods try to compute the probability of an SEU producing an error on the output of a logic gate or a bistable hit by a particle. SPICE simulations are typically performed to extract these probabilities \[24\] \[25\]. Gate level SER modeling, estimates the SER of a circuit node by computing the SEU occurrence rate probability and EPP, the error propagation probability (probability of propagating an error to an observable output). These models typically use fault injection or Binary Decision Diagrams (BDD) for SER estimation. Analytical modeling for logic level SER estimation has been done in \[26\]. The technique provides a speed-up of 4-5 orders of magnitude compared to traditional fault injection techniques. Architectural Level SER modeling, uses the Architectural Vulnerability Factor (AVF) to estimate soft error vulnerability of different blocks. Some good techniques are mentioned in \[27\].

The remainder of this chapter is organized as follows: Section 3.1 contains some definitions and background concepts that are needed for this work. A review of various techniques for SER estimation in logic circuits is presented in Section 3.2. In Section 3.3, we present techniques to obtain the Logical Soft Error Rate of combinational and sequential circuits defined at the behavioral level. A SAT-based methodology to compute the Hardware Vulnerability Factor (HVF) of various structures present in a uni-directional processor pipeline is described in Section 3.4. Finally, Section 3.5 concludes the chapter.

3.1 Background

3.1.1 Boolean Satisfiability

The Boolean satisfiability (SAT) problem is the problem of deciding if there is a truth assignment for the symbols that appear in a Boolean function such that it assigns the value True to the Boolean function. The boolean function is usually specified in product-of-sums or conjunctive normal form or CNF. A CNF is a set of clauses. This form consists of the
CHAPTER 3. SAT BASED SCHEMES FOR SER COMPUTATION

logical AND of this set of clauses. A clause is a set of literals and a literal is a variable or a negated variable. The CNF is satisfied under a given assignment for the variables if and only if (iff) all clauses are satisfied. A clause is satisfied iff at least one literal is satisfied. A literal is satisfied, iff the variable is not negated and has assigned the value 1 or the variable is negated and has assigned the value 0. A solver for boolean satisfiability must generate an input combination such that the output of the given boolean expression evaluates to 1 (SAT) or reports that the expression is unsatisfiable (UNSAT) if no such input combination exists [28].

The success of SAT solvers has led to the increased popularity of a subtle derivative, the all solution SAT solver. While in general, the SAT solvers seek to find a single solution to a SAT problem, the all-solution SAT solvers seek to find all possible solutions to a SAT problem. Typically, all-solution SAT solvers iteratively call a standard SAT solving procedure to find each solution to a problem. In each iteration, when the standard SAT solver returns a solution, a blocking clause is added to the problem to prevent it from discovering the same solution in future iterations. Since the number of solutions can be exponential to the problem size, 'compacting' the solutions at each iteration is critical for the efficiency of the solver [29].

3.2 Related Work

3.2.1 SER Estimation Techniques for combinational and sequential circuits

We present some SER estimation techniques for combinational and sequential circuits, in this section.

A study on the modeling of soft errors in combinational logic was performed at the VHDL level in [30]. This type of modeling involves several steps. First, the sensitivity of a node to a strike by an ionizing particle has to be determined by a statistical method.
Next, the generation of signals that exceed the noise margin is calculated with an analytical model. The propagation of a signal to a latch through the active paths is determined using a set of test vectors. The latching of the signal during the setup-and-hold window, resulting in a soft fault, is again described statistically. Finally, the propagation of the soft fault to the external outputs is modeled by applying a test vector set.

A tool SEAT-LA, was developed in [31] that modeled the transfer of a glitch across the combinational logic. It utilized analytical equations to model the propagation of a voltage pulse to the input of a state element. The pulse propagation was computed from each node to the output. Then the pulse-width and amplitude values were used to obtain the corresponding timing window. This analysis was done for each path of every node. The SER was obtained by summing up the timing window values for all nodes in the path.

Error Propagation Probability (EPP) of the gates has been used in [25] [32] to measure the contribution of each gate to the overall soft error rate. A gate with a higher EPP means that a bit flip at the output of the gate is more likely to cause an error at the primary outputs of the circuit. Analytical modeling for logic level SER estimation has been done in [26]. The technique provides a speed-up of 4 - 5 orders of magnitude compared to traditional fault injection techniques.

FASER was developed in [33] that used Binary Decision Diagrams (BDD) to accurately estimate the SER for cell based designs. It is a static SER analysis methodology in that it relies on the implicit enumeration of the input vector space. The algorithm formally encodes and propagates the error pulses using binary decision diagrams. By propagating the fault-encoding function to the primary outputs the algorithm can accurately predict output error probabilities. The error propagates only if the path from a fault-site to the output is sensitizable under the specific assignment of side inputs to the gates. Results for several combinational benchmarks were reported.

In [34] it was demonstrated that it is possible to analytically determine a Timing Vulnerability Factor for sequential circuit elements such as latches and SRAMs. By analyzing
the timing characteristics of each device, and the delay characteristics of the input network
to the device, the authors found that a particle strike would not affect device operation
during a substantial fraction of each cycle. For many devices, the vulnerable timing win-
dow is only 25% of the full cycle time. Until that point, typical simulations assumed that
a latch, for example, was vulnerable for 50% of each cycle; as a result, these simulations
would significantly overestimate the actual fault rate of a device.

3.3 SAT based methodology for SER Computation in Com-
binational and Sequential Circuits

In this section, a framework to accurately obtain soft error rate (SER) for high-level (be-
havioral) descriptions (Verilog or VHDL) of combinational and sequential circuits in early
design stages is presented. The SER problem is transformed into equivalent Boolean satisfi-
ability problem and state-of-the-art SAT-solvers are used to obtain SER. An automated flow
to convert high-level hardware descriptions into SAT formulations for exact SER computa-
tion has also been developed. The proposed technique has been compared with traditional
fault simulation techniques for both combinational and sequential circuits.

Obtaining SER at higher design levels is more challenging. Such designs are generally
described at behavioral level. At this stage, the modules are not yet synthesized and
computing Error Propagation Probability for example, is not possible. Moreover, the error
propagation rules for logic gates cannot easily be extended to behavioral constructs and
operations which are normally used in the high-level description of designs. Therefore, the
only existing viable solution for SER estimation of high-level descriptions is fault injection
based on random vector simulation (fault simulation) \[35\]. However, this approach has it
own shortcomings: fault simulation could be very time consuming for large designs with
many primary input bits and sequential states. Also, the accuracy of fault simulation
quickly drops as the ratio of simulated samples over the total vector space decreases.
CHAPTER 3. SAT BASED SCHEMES FOR SER COMPUTATION

Not all particle strikes result in an observable failure in the system. The effect of injected charge can be masked at multiple levels such as electrical masking, timing-window masking, logical masking, and architectural masking [36]. In this thesis, we focus on logical masking computation since the required information for electrical and timing masking computation is not available at behavioral level. Hence SER referred to in this work is essentially Logical Soft Error Rate. In order to model transient faults, we have injected bit-flips for a single clock cycle in the behavioral descriptions. Fault sites are wires for combinational circuits and registers for sequential circuits.

In this section, a technique is presented using which the Logical Soft Error Rate of combinational and sequential circuits defined at the behavioral level, can be obtained. The focus is specifically on logical masking computation since the required information for electrical and timing masking computation is not available at Register Transfer Level (RTL). Hence SER referred to in this section is essentially Logical Soft Error Rate.

3.3.1 Combinational Circuits

For combinational circuits, a particle strike at the input of a transistor, may alter the state of the transistor and introduce a bit-flip on one of the signal lines. We assume that an SEU results in a single-bit transient on one of the wires. A bit-flip causes an error if the effect of the bit-flip is observable at the primary output of the circuit. Given a bit-flip occurring on a wire, the Error Propagation Probability (EPP) for this element is a ratio between the number of input combinations which resulted in an error and the total number of possible input combinations. The SER of the circuit is calculated as the average of per-element SER for all potential error sites, i.e. all wires in the circuit. The algorithm used is shown in Figure 3.1 and is described below.

In the proposed methodology, the digital circuit is modeled as a SAT instance. We take the description of the circuit at the behavioral level (RT-level) as the input. A faulty version and a fault-free version of the circuit are instantiated. In the faulty version, a fault
CHAPTER 3. SAT BASED SCHEMES FOR SER COMPUTATION

INPUT: circuit described in behavioral HDL
OUTPUT: SER_G for the circuit
ALGORITHM:
FIND_SER (Circuit G)
For each wire (error site) w in G do
   Make another copy of the module under test.
   XOR w with a 1 in the error-inserted module
   XOR the corresponding outputs of the two modules
   in the top-level module
   OR all the outputs together to get a single output.
   Convert this description into a CNF equivalent,
   testing for the property output = 1
   Send the circuit to all-solution SAT solver
   Record the number of satisfying assignments \( N_{SAT} \)
   Compute \( EPP_w \) for wire w = \( N_{SAT} / 2^{\text{inputs}} \)
Find \( SER_G \) using the formula
\[
SER_G = \frac{\sum_{\text{fault wires}} EPP_w}{\text{Total number of wires}} \times FIT_{\text{gate}}
\]

Figure 3.1: Algorithm for the SAT-based SER Computation Methodology
is inserted by providing a statement in the HDL description in order to flip a bit on the
error site. The error sites are the variables present in the circuit description. The outputs of
fault-free design and the fault-inserted version are compared by XORing the corresponding
outputs of these two version. The output of each XOR gate is 1 if and only if the error
due to a bit-flip in the error site is propagated to that primary output. All the outputs are
ORed together to generate a miter output, i.e. the error is propagated to the output if it
appears on at least one primary output. This is shown in Figure 3.2.

Figure 3.2: Fault injection in a sample circuit
The overall description (including fault-free and fault-inserted versions, XORs, and OR gate) is then converted into a SAT instance. This SAT instance is only satisfiable, iff an input pattern is found that yields a wrong output value (1) in the presence of a fault.

The EPP due to a single bit $i$ is obtained as follows:

$$EPP_i = \frac{\text{no of SAT instances}}{2 \times \text{no of PIs}}$$

The SER of the circuit is found by summing the EPPs of all the error sites and averaging them over the number of error sites. As most SAT solvers require the circuit to be in CNF form, the model is converted to its CNF equivalent. An all-solution SAT solver is used to find the number of input assignments which satisfies the property of the output equal to 1.

**An Example**

Consider the behavioral description of a 4-bit subtracter in Verilog HDL given in Figure 3.3.

In Figure 3.3, module 4bitsub is the original circuit. Suppose a bit-flip is required to be injected in wire $D[1]$. Another copy of the module (4bitsubfaulty) is created in which the faulty bit is XORed with a 1. The outputs of faulty and fault-free modules are then XORed in module main. The single bit output out, is the final output of the circuit. If this output is 1, it means that the fault has propagated to the primary outputs.

### 3.3.2 Sequential Circuits

In sequential circuits, the bit-flip needs to be inserted for only one clock cycle. Unlike combinational circuits, this bit-flip will not be observable in the same cycle. The methodology used for sequential circuits is similar to the one described above. The difference lies in converting the circuit to its CNF equivalent. In sequential circuits, a fault on one line may take several cycles in order to be observable at the primary outputs because of memory elements. As CNF form is purely combinational, multiple unrollings have to be performed
//The fault free module
module sub4b (A,B,C)
  input [3:0] A,B;
  output [3:0] C;
  wire [3:0] D;
  assign D = A – B;
  assign C = D;
endmodule;

//The faulty module
module sub4bfaulty (A,B,C)
  input [3:0] A,B;
  output [3:0] C;
  wire [3:0] D;
  wire [3:0] temp; //additional variable to store the bit flip
  assign D = A – B;
  assign temp[0] = D[0];
  assign temp[2] = D[2];
  assign temp[3] = D[3];
  assign C = temp;
endmodule;

//The main module instantiating faulty and fault free modules
module main(A,B,out)
  input [3:0] A,B;
  output out;
  wire [3:0] temp,D,Dmod;
  sub4b sub4b_1(A,B,D);
  sub4bfaulty sub4b_2(A,B,Dmod);
  assign temp[0] = D[0]^Dmod[0]; //XOR corresponding outputs
endmodule;

Figure 3.3: Fault injection in a 4-bit subtracter
to convert the circuit to a combinational one. The error sites for sequential circuits are the outputs of all flip-flops.

![Sequential Circuit](image)

Figure 3.4: (a) A sequential circuit (b) Unrolled into $n$ combinational copies

In this case, the sequential circuit is converted into a combinational CNF by unrolling (i.e. copying) the combinational core of the sequential circuit $n$ times. This is shown in Figure 3.4(a). The bit flip is inserted in one (first) combinational copy, as explained in Sec. 3.3.1 and the corresponding outputs of the fault-free and fault-inserted versions are XORed for all $n$ unrolled combinational copies (Figure 3.4(b)). The reason for inserting the bit-flip in the first copy (cycle) is that the behavior of fault-free and faulty versions are exactly the same before the bit-flip cycle and we do not need to consider preceding cycles (i.e. the error propagation starts from the cycle in which bit-flip first occurred). Unrolling increases the size of the CNF and the SAT solvers take more time to find all possible satisfying input combinations. Also, the input space for sequential circuits will be much larger compared to combinational ones. The EPP due to a single bit $i$ is computed as follows:

$$EPP_i = \frac{\text{number of SAT instances}}{2 \times \text{number of PI}s \times \text{loop unrollings}}$$
The key issue in this approach is to unroll the circuit for a right amount of cycles \( (n) \). If the combinational core of the sequential circuit is unrolled for \( n \) cycles after bit flip, the following scenarios might happen:

1. **Error detection:** The effect of bit-flip (error) is propagated to at least one of the POs within any of \( n \) cycles after bit flip \( (Z_0-Z_n) \). In this case, at least one of the corresponding POs of faulty and fault-free copies in one of the \( n \) unrollings differ.

2. **Error masking:** The effect of bit-flip is completely masked within the \( n \) cycles. This means that the error is not detected (case 1 has not occurred) and the corresponding state outputs of the faulty and fault-free copies of the \( n^{th} \) unrolling \( (y_n) \) are exactly the same.

3. **Potential error:** In this case, the error has not propagated to POs within \( n \) cycles but exists in the circuit state. This means that the error is not detected but the corresponding state outputs of the faulty and fault-free copies of the \( n^{th} \) unrolling differ (case 1 and 2 have not occurred). In this situation, more unrollings are required to determine error detection or masking.

The procedure used to compute the SER for a sequential circuit is given in Figure 3.5 and is detailed below.

**Step A.** The circuit is initially unrolled for \( n \) time frames, where \( n \) is the sequential depth of the circuit. The bit flip is inserted in one combinational copy and the corresponding outputs are XORed for all \( n \) unrolled combinational copies. A single output is derived by ORing together all XORed outputs. The circuit is converted to its CNF equivalent and sent to an all-solution SAT solver which gives the number of satisfying assignments for which the output is 1. If the CNF instance is satisfiable, the procedure for this error site is concluded.

**Step B.** In case the CNF formula in not satisfied (case 1 not happened), we XOR the values at the \( n^{th} \) copy of the state elements to generate a new CNF formula. If the formula is still not satisfiable it means that the error has been masked (case 2). On the other hand,
$n \leftarrow \text{Sequential Depth}$

For each flip flop $f$

$flag \leftarrow 0$

While ($flag == 0$)

- Unroll the circuit for $n$ copies
- Make another copy of the unrolled module
- XOR $f$ with a 1 in the faulty module
- XOR the faulty and fault free outputs of the $n$ copies
- OR all the outputs together to get a single output
- Convert it to CNF, testing for output = 1
- Run the all-solution SAT solver on this instance

if SAT then /* error is detected */

$EPP(f) \leftarrow N_{SAT}/2^{n \times PI}$

$flag \leftarrow 1$

else

- Add clauses for XORing the states in $n^{th}$ copies
- Run the all-solution SAT solver
- if SAT then /* potential error: more unrolling */

  $n \leftarrow n + 1$

- if ($n > 2 \times numFF$) then

  $EPP(f) \leftarrow 0$

  $flag \leftarrow 1$

else /* error is masked */

  $EPP(f) \leftarrow 0$

  $flag \leftarrow 1$

Figure 3.5: Procedure for computing SER in Sequential Circuits
if the values on the state elements in faulty and fault free copies are different (i.e. the new CNF is SAT), it means that the induced error is still in the system and may appear on the outputs in a later cycle. In this case, we keep on unrolling the circuit \( n \leftarrow n + 1 \), goto Step A), until either the error propagates to a PO or the unrollings reach a maximum value which is \( 2 \times \text{numberof}FF \) in this work.

It is important to note that the SER obtained is exact for combinational circuits and very accurate for sequential circuits. The methodology models the circuit as an instance of SAT. There is a one-to-one correspondence between the circuit description and its CNF equivalent. The number of unrollings in the sequential circuits is incrementally adjusted to guarantee either error detection or masking. Also, the number of satisfying input assignments generated by the all-solution SAT solver is the exact number of input combinations that propagate the error from the error site to primary outputs. This is because the all-solution SAT solver provides the exact number of all satisfying instances to the CNF [37]. The only possible source of inaccuracy is where incremental unrolling exceeds the threshold (error is still in the system states but not propagated to primary outputs). However, this case did not occur for the benchmark circuits in our experiments. This is in contrast to fault simulation methods that approximate the SER and their accuracies are not guaranteed.

### 3.3.3 Advantages of the methodology

The approach described above has several benefits.

- Firstly, it works for all specification levels. Since its important to predict the SER early in the design cycle, this approach can be effectively used at the behavioral description level. However, it is general enough to be applicable at lower levels. In higher-level descriptions, the error sites are registers and wires. However in gate-level designs, the error sites are inputs and outputs of logic gates and flip-flops. There are tools available which can convert circuits described in HDLs (at different abstraction levels) to their CNF equivalents. We discuss some of these in the next section.
• Secondly, the SER obtained is *exact*, i.e. the accuracy of the computed SER values is always 100%. The methodology models the circuit as an instance of SAT. There is a one-to-one correspondence between the circuit description and its CNF equivalent. The number of unrollings in the sequential circuits is incrementally adjusted to guarantee either error detection or masking. Also, the number of satisfying input assignments generated by the all-solution SAT solver is the exact number of input combinations that propagate the error from the error site to primary outputs. This is because the all-solution SAT solver provides the *exact* number of all satisfying instances to the CNF [37]. This is in contrast to fault simulation methods that approximate the SER and their accuracies are not guaranteed.

• Thirdly, the approach is much more efficient than traditional fault simulation schemes. This is because of the use of SAT solvers. Recent advances in design and implementation of SAT solvers has improved the run times significantly. We will discuss this in more details in Section 3.3.5.

• Fourthly, this methodology can be applied to designs specified using different HDL languages as long as there is a tool converting specifications in that HDL to corresponding CNF.

### 3.3.4 Implementation

The methodology presented in the previous section was completely automated using C++. The flowchart is given in Figure 3.6.

Verilog HDL was used as the input language. The flow is implemented for both combinational and sequential designs described in high-level Verilog.
Combinational Designs

We constructed a fault insertion vector with each bit corresponding to an error site. All variables were XORRed with this vector. For each error site, the bit corresponding to that site was changed to a 1, while the remaining bits were forced to a 0. This 'walking-1 vector' ensured that only a single bit flipped at a time. This generated a combined description of the circuit which included a fault free module, a faulty module (containing one of the variables XORRed to a 1), a comparator at the top-level module and an OR gate to get a single output circuit.

In order to convert to its equivalent CNF, two intermediate formats were used. The Cadence SMV tool was first employed to convert verilog to SMV format. The tools in AIGER library were then used to convert from SMV format to CNF while checking for the property (output = 1). The CNF file thus obtained was sent to an all solutions SAT-solver RELSAT. RELSAT finds the number of all satisfying input solutions to a given instance. The program uses the formula for finding EPP as given in the previous section to compute SER of the circuit. Additionally, the EPP was calculated using a Verilog simulator, Veriwell for comparison purposes.
Fault simulation was also implemented, as described in Figure 3.7. A testbench was created for each circuit. All possible input combinations were applied through this testbench. The testbench along with the verilog file obtained after inserting the fault was sent to a Verilog simulator Veriwell [40]. The simulator incremented a counter whenever the output became 1. The count at the end of simulation was divided by the number of possible input combinations to get SER.

**Sequential Designs**

For sequential designs, incremental unrolling starting from sequential depth was performed to obtain the (combinational) CNF, as explained in Sec. 3.3.2. The sequential (high-level) design in Verilog was first converted into the BLIF format. The unrolling, error insertion, and XORing were performed on the BLIF format and equivalent CNF was obtained. The rest of the flow was similar to the flow for the combinational designs. The error sites considered here were register bits (flip-flops) in the design.

Since it is impossible for perform exhaustive fault simulation for sequential circuits due to
excessively large vector space after unrolling, only a fixed number of random vectors (100K, 
1M, or 10M) were simulated. For instance, a small sequential design with 10 primary inputs 
and sequential depth of 5 requires $2^{10 \times 5} = 2^{50}$ vectors for an exhaustive fault simulation. 
We however performed exhaustive fault simulation for smaller circuits. In this case, the 
number of vectors needed for exhaustive simulation was less than 10 million.

### 3.3.5 Experimental Results

The tool was developed in C++, and experiments were carried out on several benchmark 
circuits given in behavioral verilog. The simulations were carried out on a 65 node Linux 
Cluster. Each node contained two Xeon processors and 4 GB of RAM. The nodes were 
interconnected with Gigabit Ethernet. For comparison purposes, the EPP for every bit was 
calculated using the fault simulation flow as well. The signal probability of the output for 
the miter circuit was a measure of the EPP of a single bit.

#### Combinational Benchmarks

A high-level simulator Veriwell [40] was used to perform fault simulations. The timing 
comparisons are given in Table 3.1. Since the SAT-based flow is an exact method (100% 
accuracy), the fault simulation is performed exhaustively to make a fair comparision.

<table>
<thead>
<tr>
<th>circuit</th>
<th>error sites</th>
<th>$time_{SAT}$ (sec)</th>
<th>$time_{FS}$ (sec)</th>
<th>speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>c432</td>
<td>235</td>
<td>266.80</td>
<td>4278.20</td>
<td>16.04</td>
</tr>
<tr>
<td>c499</td>
<td>81</td>
<td>46.80</td>
<td>682.10</td>
<td>14.57</td>
</tr>
<tr>
<td>74181</td>
<td>52</td>
<td>9.36</td>
<td>98.80</td>
<td>10.56</td>
</tr>
<tr>
<td>74182</td>
<td>9</td>
<td>0.22</td>
<td>1.80</td>
<td>8.18</td>
</tr>
<tr>
<td>74L85</td>
<td>21</td>
<td>8.68</td>
<td>90.40</td>
<td>10.41</td>
</tr>
<tr>
<td>74283</td>
<td>14</td>
<td>0.27</td>
<td>2.10</td>
<td>7.78</td>
</tr>
<tr>
<td>fadd8</td>
<td>26</td>
<td>4.10</td>
<td>39.50</td>
<td>9.63</td>
</tr>
<tr>
<td>fadd16</td>
<td>50</td>
<td>9.88</td>
<td>122.70</td>
<td>12.42</td>
</tr>
<tr>
<td>fadd32</td>
<td>82</td>
<td>21.40</td>
<td>284.20</td>
<td>13.28</td>
</tr>
<tr>
<td>Average</td>
<td></td>
<td>40.83</td>
<td>622.20</td>
<td>15.24</td>
</tr>
</tbody>
</table>
In Table 3.1, Column 2 shows the number of fault sites present in the circuit. In Column 3, we present the time taken to obtain the SER of the circuit using SAT solver. The time taken for exhaustive fault simulation is shown in Column 4. Finally, Speedup achieved for each circuit is given in Column 5. The average speedup obtained for the benchmarks was also computed. This is given in the last row of Table 3.1. The results obtained from the SAT-based flow matched exactly with the exhaustive fault simulation flow for all error sites and all benchmark circuits.

The first two circuits in Table 3.1 are ISCAS85 benchmark circuits. The next 4 circuits are behavioral descriptions of MSI components in the 74xxx series. fadd8, fadd16 and fadd32 are extensions to the behavioral descriptions of a fast adder obtained from [41]. Circuits with medium complexity were used here because of the limitation of the exponential run time for the exhaustive fault simulation. The error sites are the number of inputs and variables present in the behavioral description of the circuit. Note that performing exhaustive fault simulation is intractable for large circuits. Since All-SAT solvers find all possible solutions to the instance, the approach used is similar to exhaustive test simulation.

Time taken by all-solution SAT-solver RELSAT [37] is proportional to the size of the CNF file. The CNF file size in turn is dependent upon the property to test for a particular circuit. Hence, it varies greatly with which fault site is under consideration. On the other hand, a fault simulator, simulates the test vectors provided (exhaustive in this case) for the whole circuit. Therefore, the time taken by a fault simulator is almost the same for all fault sites in the circuit. It is mainly a function of the number of simulated input patterns which in turn is exponential to the number of primary inputs in exhaustive simulation.

It can be observed from Table 3.1 that the time taken by RELSAT is an order of magnitude lower than that taken by fault simulation for most of the circuits. For larger circuits, the difference is slightly less than two orders of magnitude. In other words, the speed-up of the SAT-based flow is greater for larger circuits.
Sequential Benchmarks

We performed fault simulations for sequential circuits using SIS tools \cite{sis}. Since exhaustive fault simulation was impractical for unrolled version of most sequential circuits, we performed fault simulation on these benchmarks with 100K, 1M and 10M randomly generated vectors. The random number generator was seeded with the same value in all three cases to ensure that 100K vectors are a subset of 1M vectors and 1M vectors are a subset of 10M vectors.

Validation using Smaller Circuits

Table 3.2: Validation results using small-sized sequential circuits

<table>
<thead>
<tr>
<th>circuit</th>
<th>PIs</th>
<th>FFs</th>
<th>SER</th>
<th>$\text{time}_{\text{SAT}}$ (sec)</th>
<th>$\text{time}_{\text{FS}}$ (sec)</th>
<th>speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>mc</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1.32</td>
<td>15.67</td>
<td>11.87</td>
</tr>
<tr>
<td>shiftreg</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>1.24</td>
<td>6.04</td>
<td>4.85</td>
</tr>
<tr>
<td>lion</td>
<td>2</td>
<td>2</td>
<td>0.485</td>
<td>0.66</td>
<td>2.96</td>
<td>4.48</td>
</tr>
<tr>
<td>train</td>
<td>1</td>
<td>2</td>
<td>0.281</td>
<td>0.95</td>
<td>2.70</td>
<td>2.84</td>
</tr>
<tr>
<td>ex3</td>
<td>2</td>
<td>4</td>
<td>0.730</td>
<td>37.40</td>
<td>498.34</td>
<td>13.18</td>
</tr>
<tr>
<td>bbtas</td>
<td>2</td>
<td>3</td>
<td>0.696</td>
<td>3.87</td>
<td>40.35</td>
<td>10.42</td>
</tr>
<tr>
<td>s27</td>
<td>3</td>
<td>3</td>
<td>0.230</td>
<td>3.88</td>
<td>42.83</td>
<td>11.03</td>
</tr>
<tr>
<td>dk14</td>
<td>3</td>
<td>3</td>
<td>0.568</td>
<td>149.62</td>
<td>1425.06</td>
<td>9.52</td>
</tr>
</tbody>
</table>

In order to verify the correctness of our approach for sequential circuits, we did exhaustive fault simulation for small MCNC benchmark circuits. The results are presented in Table 3.2.

In Table 3.2, Column 2 shows the number of Primary Inputs (PIs) and Column 3 shows the number of flip-flops (FFs) in the circuits. The circuits were unrolled for $1.5 \times \text{no. of FFs}$. Column 4 present the SER obtained by presented technique and the exhaustive fault simulation (in all cases, the numbers matched exactly). As mentioned in the previous section, the fault sites are the output of flip-flops. In Columns 5 and 6, we report the time taken by the SAT Solver and exhaustive fault simulation to obtain the SER. Finally, the speed up of our approach over exhaustive fault simulation is presented in the last column.
These experiments confirmed that the results obtained from SAT-based flow match exactly with the exhaustive fault simulation flow for all error sites, i.e. the SAT-based flow is 100% accurate.

**Comparison using Larger Circuits** We also performed simulations on high level descriptions of larger benchmark circuits and the results are presented in Table 3.3. The first seven circuits in Table 3.3 are ISCAS89 benchmark circuits while the last three circuits are obtained from VIS benchmark suite. We initially unrolled the circuits for the number of time frames equal to the sequential depth. In case the error did not propagate to the POs but remained on the sequential elements, further unrolling was performed. We continued unrolling until one of the following was satisfied: 1) error masking, 2) error detection, or 3) the number of unrollings became greater than \(2 \times \text{number of FFs}\).

Table 3.3 shows the characteristics of the benchmarks. It can be seen that four of the ten benchmarks required further unrollings. The number of unrollings were less than threshold and very close to the sequential depth. Since case 3 did not occur for any of the circuits, the results were 100% accurate. The SER obtained and the time taken by the SAT solver is also shown.

Since exhaustive fault simulation is intractable on these circuits, we simulated the circuit with 100k, 1M and 10M randomly generated vectors. For each of these group of vectors we report the SER values obtained from random fault simulation, the percentage difference between the actual SER (obtained from the SAT flow) and the SER obtained by fault simulating this group of vectors, the time spent in fault simulation, and the overhead (as the ratio of corresponding run time) over SAT-based flow.

The results are presented only for these circuit since the fault simulation flow was completely intractable for larger circuits. These results show that random fault simulation extremely under-estimates SER values. For these circuits, the runtime of 100K vector simulation is comparable with SAT flow although the fault simulation under-estimates the
Table 3.3: Comparison of SAT-flow and random fault simulation for sequential circuits

<table>
<thead>
<tr>
<th>circuit</th>
<th>name</th>
<th>seq depth</th>
<th>unrolling</th>
<th>PIs</th>
<th>FFs</th>
<th>SER $\times 10^{-3}$</th>
<th>time</th>
<th>SAT Flow</th>
<th>name</th>
<th>seq depth</th>
<th>unrolling</th>
<th>PIs</th>
<th>FFs</th>
<th>SER $\times 10^{-3}$</th>
<th>time</th>
<th>runtime</th>
<th>Inacc.</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>s298</td>
<td>8</td>
<td>8</td>
<td>3</td>
<td>14</td>
<td>7.796</td>
<td>659.58</td>
<td></td>
<td>s298</td>
<td>89</td>
<td>9</td>
<td>15</td>
<td>0.327</td>
<td>1311.24</td>
<td>331.03</td>
<td>6</td>
<td>91.43</td>
<td>10.25x</td>
</tr>
<tr>
<td></td>
<td>s344</td>
<td>6</td>
<td>8</td>
<td>9</td>
<td>15</td>
<td>0.324</td>
<td>1324.40</td>
<td></td>
<td>s344</td>
<td>69</td>
<td>9</td>
<td>15</td>
<td>0.311</td>
<td>737.49</td>
<td>48.62</td>
<td>8</td>
<td>91.43</td>
<td>10.13x</td>
</tr>
<tr>
<td></td>
<td>s349</td>
<td>6</td>
<td>8</td>
<td>9</td>
<td>15</td>
<td>0.977</td>
<td>479.42</td>
<td></td>
<td>s349</td>
<td>69</td>
<td>9</td>
<td>15</td>
<td>0.316</td>
<td>783.05</td>
<td>49.37</td>
<td>8</td>
<td>91.43</td>
<td>10.14x</td>
</tr>
<tr>
<td></td>
<td>s382</td>
<td>11</td>
<td>11</td>
<td>3</td>
<td>21</td>
<td>0.327</td>
<td>1324.40</td>
<td></td>
<td>s382</td>
<td>11</td>
<td>11</td>
<td>3</td>
<td>21</td>
<td>0.324</td>
<td>1311.24</td>
<td>6</td>
<td>91.43</td>
<td>10.13x</td>
</tr>
<tr>
<td></td>
<td>s386</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>6</td>
<td>0.977</td>
<td>479.42</td>
<td></td>
<td>s386</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>6</td>
<td>0.977</td>
<td>479.42</td>
<td>6</td>
<td>91.43</td>
<td>10.14x</td>
</tr>
<tr>
<td></td>
<td>s444</td>
<td>11</td>
<td>11</td>
<td>3</td>
<td>21</td>
<td>0.324</td>
<td>1324.40</td>
<td></td>
<td>s444</td>
<td>11</td>
<td>11</td>
<td>3</td>
<td>21</td>
<td>0.324</td>
<td>1311.24</td>
<td>6</td>
<td>91.43</td>
<td>10.15x</td>
</tr>
<tr>
<td></td>
<td>s510</td>
<td>2</td>
<td>4</td>
<td>19</td>
<td>6</td>
<td>0.161</td>
<td>84.63</td>
<td></td>
<td>s510</td>
<td>2</td>
<td>4</td>
<td>19</td>
<td>6</td>
<td>0.161</td>
<td>84.63</td>
<td>6</td>
<td>91.43</td>
<td>10.12x</td>
</tr>
<tr>
<td></td>
<td>s1</td>
<td>5</td>
<td>5</td>
<td>8</td>
<td>5</td>
<td>0.066</td>
<td>691.66</td>
<td></td>
<td>s1</td>
<td>5</td>
<td>5</td>
<td>8</td>
<td>5</td>
<td>0.066</td>
<td>691.66</td>
<td>6</td>
<td>91.43</td>
<td>10.16x</td>
</tr>
<tr>
<td></td>
<td>traffic</td>
<td>10</td>
<td>10</td>
<td>3</td>
<td>10</td>
<td>0.140</td>
<td>317.97</td>
<td></td>
<td>traffic</td>
<td>10</td>
<td>10</td>
<td>3</td>
<td>10</td>
<td>0.140</td>
<td>317.97</td>
<td>6</td>
<td>91.43</td>
<td>10.25x</td>
</tr>
<tr>
<td></td>
<td>minmax</td>
<td>6</td>
<td>6</td>
<td>5</td>
<td>6</td>
<td>0.087</td>
<td>580.05</td>
<td></td>
<td>minmax</td>
<td>6</td>
<td>6</td>
<td>5</td>
<td>6</td>
<td>0.087</td>
<td>580.05</td>
<td>6</td>
<td>91.43</td>
<td>10.25x</td>
</tr>
<tr>
<td></td>
<td>average</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>696.949</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>696.949</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>circuit</th>
<th>100K Vector Random FS</th>
<th>1M Random Vector FS</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>SER $\times 10^{-3}$</td>
<td>time sec</td>
</tr>
<tr>
<td>s298</td>
<td>2.611</td>
<td>648.70</td>
</tr>
<tr>
<td>s344</td>
<td>0.028</td>
<td>1298.82</td>
</tr>
<tr>
<td>s349</td>
<td>0.033</td>
<td>1316.44</td>
</tr>
<tr>
<td>s382</td>
<td>0.061</td>
<td>744.28</td>
</tr>
<tr>
<td>s386</td>
<td>0.197</td>
<td>502.63</td>
</tr>
<tr>
<td>s444</td>
<td>0.061</td>
<td>823.28</td>
</tr>
<tr>
<td>s510</td>
<td>0.013</td>
<td>91.43</td>
</tr>
<tr>
<td>s1</td>
<td>0.014</td>
<td>702.66</td>
</tr>
<tr>
<td>traffic</td>
<td>0.027</td>
<td>309.54</td>
</tr>
<tr>
<td>minmax</td>
<td>0.017</td>
<td>642.28</td>
</tr>
<tr>
<td>average</td>
<td></td>
<td>708.01</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>circuit</th>
<th>10M Random Vector FS</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>SER $\times 10^{-3}$</td>
</tr>
<tr>
<td>s298</td>
<td>6.573</td>
</tr>
<tr>
<td>s344</td>
<td>0.238</td>
</tr>
<tr>
<td>s349</td>
<td>0.228</td>
</tr>
<tr>
<td>s382</td>
<td>0.251</td>
</tr>
<tr>
<td>s386</td>
<td>0.771</td>
</tr>
<tr>
<td>s444</td>
<td>0.255</td>
</tr>
<tr>
<td>s510</td>
<td>0.115</td>
</tr>
<tr>
<td>s1</td>
<td>0.053</td>
</tr>
<tr>
<td>traffic</td>
<td>0.116</td>
</tr>
<tr>
<td>minmax</td>
<td>0.073</td>
</tr>
<tr>
<td>average</td>
<td></td>
</tr>
</tbody>
</table>
CHAPTER 3. SAT BASED SCHEMES FOR SER COMPUTATION

SER values by over 80%. Even for 10M vector simulation which is 102 times more time-consuming than SAT-based flow, the estimated SER values are, on average, 21% lower than the actual values. These results clearly show that fault simulation is not a proper approach for SER estimation of sequential circuits and the estimated values are significantly lower than actual values. In other words, if the reliability parameters of a design are obtained using fault simulation approaches, the actual reliability requirements may not be met and this may result in catastrophes in critical applications.

3.3.6 Scalability of the SAT-based Technique

In order to demonstrate the scalability of the proposed technique, we applied it to large benchmark circuits. The results are presented in Table 3.4. The first eight circuits in Table 3.3 are ISCAS 89 benchmark circuits with the largest one having more than 1,700 flip flops. The last two circuits are RISC processors cores at RT-level. The gate count is taken from the intermediate BLIF description, where gates can be multiple-input (> 2) NAND and NOR gates. The average number of clauses obtained after unrolling the circuit and average time taken by the SAT solver is reported in the last two columns. The time reported is relative to benchmark s444 whose results are reported in Table 3.3. It can be observed from the results that the technique scales well for large sequential circuits.

<table>
<thead>
<tr>
<th>circuit</th>
<th>FFs</th>
<th>gates</th>
<th>clauses</th>
<th>relative time</th>
</tr>
</thead>
<tbody>
<tr>
<td>s444</td>
<td>21</td>
<td>181</td>
<td>38,339</td>
<td>1</td>
</tr>
<tr>
<td>s5378</td>
<td>164</td>
<td>2,779</td>
<td>110,124</td>
<td>2.91</td>
</tr>
<tr>
<td>s9234</td>
<td>211</td>
<td>5,597</td>
<td>329,779</td>
<td>7.38</td>
</tr>
<tr>
<td>s13207</td>
<td>669</td>
<td>8,027</td>
<td>238,929</td>
<td>6.71</td>
</tr>
<tr>
<td>s15850</td>
<td>597</td>
<td>9,786</td>
<td>476,267</td>
<td>12.65</td>
</tr>
<tr>
<td>s35932</td>
<td>1,728</td>
<td>16,353</td>
<td>575,664</td>
<td>15.13</td>
</tr>
<tr>
<td>s38417</td>
<td>1,636</td>
<td>22,397</td>
<td>633,478</td>
<td>16.91</td>
</tr>
<tr>
<td>s38584</td>
<td>1,452</td>
<td>19,407</td>
<td>630,918</td>
<td>16.48</td>
</tr>
<tr>
<td>frisc</td>
<td>932</td>
<td>6,743</td>
<td>203,852</td>
<td>5.53</td>
</tr>
<tr>
<td>r4000</td>
<td>249</td>
<td>1,357</td>
<td>89,559</td>
<td>2.51</td>
</tr>
</tbody>
</table>

Table 3.4: Experiments for Large sequential circuits
It can be inferred from the results that modeling soft errors using SAT solvers is a viable, accurate, and time-efficient approach compared to fault simulation. Since, exhaustive fault simulation is intractable for even medium sized sequential circuits, SAT based SER computation provides an accurate and feasible solution. The methodology presented in this section, has been extended to obtain a similar metric for in-order processor pipelines. The details and experimental results are given in Section 3.4.

3.4 SAT-based Computation of Hardware Vulnerability Factor

In this section, we extend the concept of SAT-based SER estimation for combinational and sequential circuits to compute a metric called Hardware Vulnerability Factor (HVF). The Instruction Set Architecture (ISA) is an interface between processor organization/structure and instruction level programming. Hardware Vulnerability Factor (HVF) considers the reliability below ISA. It is the probability that an error in any bit of the internal processor structure will result in an error in a program visible state. The metric captures the reliability of a particular structure, independent of the vulnerabilities inherent to specific program running on the microprocessor. This is in contrast to the Architecture Vulnerability Factor (AVF) which is application dependent and crosses ISA boundary \[3\]. By separating application independent vulnerabilities (captured in HVF) from application related vulnerabilities, it is possible to optimize the overall reliability through hardware design and application/compiler optimizations. The Boolean Satisfiability techniques described above, have been used in this work for exact computation of HVF for various structures in a processor pipeline.
3.4.1 Computing HVF

In this section, a technique to obtain HVF of a high-level (behavioral or RTL) processor description using Boolean Satisfiability (SAT) is presented. The HVF computation problem is transformed into an equivalent Boolean satisfiability (SAT) problem and state-of-the-art SAT solvers are used to obtain HVF. This approach can obtain exact HVF values in contrast to fault simulation estimation approaches.

In order to obtain HVF, we compute error propagation probability (EPP) which is the probability of error propagated from an error site to the primary outputs. EPP of an error site (bit in the processor description) is a ratio between the number of input combinations which resulted in an observable error and the total number of possible input combinations. In the proposed SAT-based approach, the processor description (RT-level) is modeled as a SAT instance. The fault injection methodology is presented below.

A faulty version and a fault-free version of the description are instantiated. In the faulty version, a fault is inserted by providing a statement in the HDL description in order to flip a bit on the error site. The outputs of fault-free design and the fault-inserted version are compared by XORing the corresponding outputs of these two versions. The output of each XOR gate is 1 if and only if the error due to a bit-flip in the error site is propagated to that primary output. All the outputs are ORed together to generate a miter output, i.e. the error is propagated to the output if it appears on at least one primary output.

The overall description (including fault-free and fault-inserted versions, XORs, and OR gate) is then converted into a SAT instance. This SAT instance is only satisfiable, iff an input pattern is found that yields a wrong output value (1) in the presence of a fault.

In pipeline structures (sequential circuits in general), the bit-flip needs to be inserted for only one clock cycle and this bit-flip may not be observable in the same cycle. The approach mentioned in Section 3.3.2 needs to be slightly modified for pipeline structures. The difference lies in converting the circuit to its CNF equivalent. In pipelines (sequential circuits), an error on one bit may take several cycles in order to be observable at the primary
outputs because of sequential behavior of the circuits. As CNF form is purely combinational, multiple unrollings have to be performed to convert the circuit to a combinational one. The approach taken for unrolling the processor description is similar to one mentioned in Section 3.3.2.

In this case, the pipeline structure is converted into a combinational CNF by unrolling (i.e., copying) the combinational core of the sequential circuit equal to the number of pipeline stages in the processor. The bit flip is inserted in the particular combinational copy corresponding to the error site. The corresponding outputs of the fault-free and fault-inserted versions are XORed based on the pipeline structure of the processor. Specifically, the error is observable if it is propagated to architecturally visible states, which are ISA registers (register file, program counter) and the memory interface. Therefore, there is an error if the content of any ISA register is erroneous at the end of write-back stage (WB) or an incorrect value is written into the memory. The latter happens when an incorrect write-enable signal is asserted or incorrect value (address or data) is generated in the Memory Access stage (MEM). Based on the particular pipeline structure, the corresponding signals need to be XORed at particular combinational copy representing those pipeline stages.

This approach increases the size of the CNF and the SAT solvers take more time to find all possible satisfying input combinations. Also, the input space for sequential circuits will be much larger compared to combinational ones. The EPP due to a single bit $i$ is computed as follows:

$$EPP_i = \frac{\text{number of SAT instances}}{2 \times \text{number of PI}s \times \text{loop unrollings}}$$

The above algorithm is presented in Figure 3.8.
**INPUT:** Processor description in behavioral HDL

**OUTPUT:** HVF for the processor

**ALGORITHM:**

FIND_HVF (Processor P)

1. Make another copy of the module under test.
2. XOR the corresponding outputs of the two modules
   a. ISA Registers after WB stage
   b. Memory write values (data and address) after MEM (write) stage
   c. Memory enable signal at all stages
3. OR all the outputs together to get a single output.
4. For each bit (error site) \( w \) in \( P \) do
   a. XOR \( w \) with a 1 in the error-inserted module in the corresponding combinational copy
   b. Unroll the description by the number of pipeline stages to obtain a pure combinational version
   c. Convert this description into a CNF equivalent
      i. Test for the property output = 1
   d. Send the circuit to all-solution SAT solver
   e. Record the number of satisfying assignments \( (N_{\text{SAT}}) \)
   f. Compute \( EPP_w \) for wire \( w = (N_{\text{SAT}})2^{\text{inputs}} \)
5. Find HVF\(_P\) using the formula

\[
\text{HVF}_P = \frac{\sum_{w \in \text{errors}} EPP_w}{\text{Total number of bits}}
\]

Figure 3.8: SAT-based approach for HVF Computation

### 3.4.2 Implementation Details

The tool was developed in C++, and experiments were carried out on two pipeline RISC processor descriptions given in behavioral verilog. As mentioned in the previous section, the processor description is converted from behavioral verilog to CNF for sending to a all-solution SAT solver RELSAT [37]. The flow to convert to CNF is shown in Figure 3.9. Synopsys was used to get a gate-level description in Verilog for the processor. Some utilities provided with the ABC Verification Suite [43] were used to convert the description to BLIF format. AIGER utilities [39] were used to convert from BLIF to CNF.

![Figure 3.9: Flow from Behavioral Verilog to CNF](image_url)
3.4.3 Experimental Results

The experiments were carried out on two pipeline implementations of RISC processors described in Verilog at RT-level. One was a 5-stage pipeline implementation of a MIPS processor, while the other was a 6-stage implementation of a more complete processor that could handle floating point instructions as well.

Standard MIPS Processor

We have used a five-stage pipeline implementation of the 32-bit integer MIPS processor described in Verilog at RT-level. The microprocessor is interfaced with the memory. The primary inputs are 32 bits of instruction, memory address and data-in (each 32 bits), as well as control signals like reset. The outputs are memory write and data-out (each 32-bits) as well as read and write enable signals. We also put the architecturally visible registers ($0-$31) in the primary outputs. Since the combinational core is unrolled five times (because of the pipeline depth), the values of 32-bits of instruction inputs corresponding to cycle 2 to 5 are set to zero, representing no-op as the next instructions in the pipeline. This means that only one instruction is considered in the pipeline and all remaining are set to no-op. This allows us to analyze per-instruction HVF. We have also fixed the opcode for the instruction bits such that only valid MIPS instructions are used as the fault-free values (i.e. illegal instructions are avoided in fault-free inputs). After converting the description to the BLIF format, there are 1319 bits (flip-flops) which are used as the error sites. We obtained HVF for each bit in the processor structure for all 54 MIPS integer instructions. The internal bits were grouped based on major resources, namely registers on the boundaries of the ALU, pipeline registers, control FSM register, and register file. The distribution of bits in these resources is shown in Fig. 3.10(a). It can be seen that register file contains three quarters of all bits. The breakdown of overall HVF per resources is shown in Figure 3.10(b). It shows that more than 90% of overall HVF is due to errors in the register file.

We have also looked at the bits with highest HVF. By sorting all 1319 bits based on their
CHAPTER 3. SAT BASED SCHEMES FOR SER COMPUTATION

Figure 3.10: (a) Distribution of bits in different resources (b) HVF distribution for different resources

Figure 3.11: (a) distribution of top 5% bits with highest HVF (b) top 20% bits

HVF in descending order and considering those in top 5 and 20 percentile, the distribution of most vulnerable bits in major resources is shown in Fig. 3.11. Top 5% consists of 66 most vulnerable FFs and top 20% contains 264 most vulnerable FFs. These results suggest that although control unit contributes to only 5% of total HVF, it contains 20%-26% of most vulnerable bits.

Figure 3.12 shows the distribution of flip-flops (bits) based on their HVFs. This shows that majority of bits have very large HVF. This is because register file constitute the majority of bits and since they are observable outputs, their HVFs are very high. We have also performed HVF analysis for various instructions. We have grouped 54 integer
instructions into four groups: arithmetic, logical, branch/jump, and store/load. The results are shown in Fig. 3.13. These results suggest that HVF variation for different instructions is almost negligible.

![Figure 3.12: Distribution of flip-flops (bits) based on HVF](image)

**UCore Processor**

UCore is a 32-bit opencore RISC microprocessor which implements all the instructions of the MIPS32R2 Instruction Set. The processor has 6 pipeline stages: Instruction Fetch, Instruction Decode, Register Fetch, Execution, Memory Access and Write Back. It has a co-processor which implements some multimedia instructions. The primary inputs and outputs are similar to the MIPS processor described in Section 4.1. Since, the pipeline is six stages deep, the combinational core is unrolled six times. The value of the 32-bit instruction vector is set to 0 in cycles 2 to 6 representing a no-op. We applied the procedure given in the previous section and obtained a BLIF netlist consisting of 2289 flip-flops. These were marked as the error sites. The internal bits were grouped based on five resources namely ALU registers, pipeline registers, co-processor registers, control registers and the register file. The distribution of bits is shown in Fig 3.14 (a). It can be observed that the number of bits in the pipeline registers is similar to that of the register file. The overall HVF per resource is shown in Fig 3.14 (b). It shows that 61% of the overall HVF is due to the errors
in the register file although the register file constitutes only 44% of the resources.

The distribution of most vulnerable bits in the five resources is shown in Fig 3.15. Top 5% consists of 114 most vulnerable flip flops while top 20% contains 457 most vulnerable flip flops. The results suggest that although pipeline registers constitute 41% of the resources, they contribute to only 21% of the most vulnerable flip flops. The register file on the other hand contributes to 60% of the most vulnerable flip flops.

Figure 3.16 shows a histogram of the number of bits and their corresponding HVFs. As compared to the results of Figure 3.12 it can be observed that more flip-flops have HVFs in the range 0.4 – 0.8 in UCore compared to MIPS. This is because the number of bits in
Figure 3.15: (a) Distribution of top 5% bits with highest HVF for UCore processor (b) top 20% bits

Unlike fault simulation which is an estimation method, the proposed technique is an exact method (100% accurate). Using this approach, accurate values of the Hardware Vulnerability Factor can be obtained in an efficient manner.

3.5 Summary

Reliability is increasingly becoming a major concern with the technology scaling at nanoscale. In this chapter, we have demonstrated the use of a SAT solver to model soft errors in early
design stages. We compute soft error rate (SER) for combinational and sequential circuits by finding out the CNF equivalent of the circuit and sending it to a SAT-solver. Moreover, we use a similar SAT-based technique to compute the Hardware Vulnerability Factor of various structures in an inorder processor pipeline. Experiments are carried out on various benchmarks as well as two MIPS-like processor descriptions, and results are presented.
Chapter 4

Detection and Recovery of Soft Errors in Processor Pipelines

Reliability is one of the critical aspects of present-day microprocessor based systems. In General, users expect their systems to be fault free. In order to provide a significantly high level of reliability to the customers, vendors try to build systems that can tolerate malfunctions of hardware and software. These malfunctions can occur either due to permanent defects (like wear and tear and manufacturing anomalies) or due to transient errors (like electromagnetic interference, power glitches and cosmic radiations). The continuous shrinking of dimensions and operating voltages of transistors has increased the processor’s sensitivity against radiations, power glitches and other sources of transient errors. As mentioned in [45], these errors can affect the control flow of a program, change the system status or modify the data stored in memory. Further, if the system does not perform some runtime checking, an erroneous output might not be detected and be used as a correct output. Many present day digital systems are high-availability systems. They are used in a variety of fields where failures can be catastrophic. Some examples include biomedical, aerospace and banking applications where events like spontaneous reboots or incorrect results cannot be tolerated. Hence, runtime error correction and/or redundancy techniques are mandatory.
to overcome the effects of transient errors [14].

Although, storage units like RAMs and caches are mostly covered by Error Correcting Codes (ECC) in present day systems, in general, a large part of microprocessor remains unprotected from soft errors. For instance, a large part of control logic as well as structures in the front-end, remain unprotected. This is due to the fact that designing effective techniques that can replace unit-level replication is difficult. Moreover, all error detection and correction approaches introduce a penalty in performance, power, die size and design time [3]. Consequently, designers must carefully weigh the benefits of adding these techniques against their cost. Although a microprocessor with inadequate protection from transient errors, may prove useless due to its low reliability, excessive protection may make the resulting product uncompetitive in cost and/or performance. Hence, a delicate balance is needed between cost and reliability especially during the early design phases.

The remainder of this chapter, is organized as follows:

Section 4.1 provides an overview of the classic 5-stage MIPS pipeline. A literature review of various fault tolerant techniques introduced in recent years is given in Section 4.2. A scheme for transient error detection and correction based on ECC, is presented in Section 4.3. Section 4.4 contains various schemes for protecting front-end structures in modern superscalar pipelines. A summary is presented in Section 4.5.

4.1 Background

Pipelining is an implementation technique that exploits parallelism among the instructions in a sequential instruction stream. It is a key implementation technique used to make fast CPUs [20]. Pipelining increases the CPU instruction throughput - the number of instructions per unit of time. The increase in instruction throughput means that a program runs faster and has lower execution time, even though no single instruction runs faster. Most modern superscalar processors like [46] and [21] are based on MIPS-like pipelines.
Multi-core systems as mentioned in [45] also consist of simple cores that are based on MIPS multi-stage pipelines.

The classic five-stage MIPS pipeline consists of the following stages:

- **Instruction Fetch (IF):** Send the program counter (PC) to memory and fetch the current instruction from memory. Update the PC to the next sequential PC by adding 4 to the PC.

- **Instruction Decode (ID):** Decode the instruction and read the registers corresponding to register source specifiers from the register file. Check for possible branches.

- **Execution (EX):** ALU operates on the operands prepared in the previous cycle.

- **Memory Access (MA):** If the instruction is a load, memory does a read using the effective address computed in the previous cycle. For a store, the memory writes the data from the registers using the effective address.

- **Write Back (WB):** Write the result into register file.

The pipeline is depicted in Figure 4.1.

Most MIPS-based processors implement a concept known as *forwarding*. Forwarding allows the write back stage to use the values of the registers from the output of the Execute stage rather than the register files. There are situations called *hazards*, that prevent the next instruction in the stream from executing during its designated clock cycle. *Structural*
hazards arise from resource conflicts when the hardware is not able to support the simultaneous execution of instructions. Data hazards arise when an instruction depends on the results of a previous instruction that has not yet finished execution. Control hazards arise from the pipelining of branches and other instructions that change the PC. Avoiding a hazard often requires delaying some instructions in the pipeline. This is called a stall. No new instructions are fetched during a stall. Another technique used is called flush. This clears all the instructions in the pipeline and is needed in case an exception occurs. Most modern processors are equipped with hazard detection logic. This logic checks for potential hazards and overcomes them using stall and flush.

4.1.1 Architecture of Superscalar Pipelines

In this section, we describe the core structures and stages within the pipeline of a superscalar processor. Although most of the elements are generic, it closely resembles the Alpha 21264 processor [47] [48].

Our baseline processor is a highly out-of-order, superscalar implementation of the Alpha architecture [48], using many advanced architectural features like multiple issues per cycle, speculative execution etc. The architecture is similar to other state-of-the-art superscalar processors and our proposed techniques are applicable to those as well.

Our pipeline consists of the following stages as mentioned in [49]:

- Instruction Fetch Stage: The processor fetches instructions from the instruction buffer in this stage. They are supplied to the rest of the pipeline and the next PC is computed.

- Decode Stage: The decode stage tries to decode number of instructions equal to its width, in every cycle. Checks for any branches and squashes in case of incorrect branch prediction.

- Register Rename Stage: The rename stage tries to rename number of instructions
equal to its width, in every cycle. It holds onto the rename history of all instructions with destination registers, storing the architected register and the new physical register in a rename table. The Rename Table is usually implemented as a Content Addressable Memory (CAM), and is mostly indexed by architectural registers. The rename stage further calculates the number of free Re-order buffer (ROB), Issue Queue (IQ) and Load Store Queue (LSQ) entries and blocks the instructions in case of no free ROB, IQ or LSQ entries.

- IEW Stage: The Issue, Execute, Writeback (IEW) stage handles the dispatching of instructions to the LSQ/IQ as part of the issue stage, and has the IQ try to issue instructions each cycle. The execute portion of IEW separates memory instructions from non-memory instructions, either telling the LSQ to execute the instruction, or executing the instruction directly. The operands are read from the register file in execute stage. At the end of execution, results are written to the register file (Write Back), and the name of updated registers are broadcasted to the Issue Queue.

- Commit Stage: It checks the head of the ROB and finds out if it is ready to commit. If it is not ready, Commit waits till the ready_to_commit flag is high. It frees the ROB (by removing the Head instruction) and the releases the physical register back into the pool of available registers.

Figure 4.2 shows the sequence that an instruction goes through from Instruction Fetch to Commit.

4.2 Related Work

4.2.1 Traditional Fault Tolerant Techniques

One of the first fault-tolerance approaches used to protect high-end computer systems was dual modular redundancy. Dual modular redundancy, employs spatial redundancy in the
CHAPTER 4. SOFT ERRORS IN PROCESSOR PIPELINES

Figure 4.2: The microarchitecture of a superscalar pipeline. (1) Instruction is fetched from Instruction Cache. (2) It is decoded. (3) Registers are renamed to overcome output dependencies. (4) Rename stage checks for free IQ / ROB entries. (5) Instructions are issued to execute on Functional Units (FU). (6) FU access the register files for operands and write back. (7) Instructions are committed when they reach Head of the ROB. (8) Upon commit, the entry in the rename table, corresponding to the destination register is removed.

form of two microprocessors operating in lockstep. The output of the two microprocessors is checked by an external checker. If any deviation at the output of the two microprocessors is detected, a system error is flagged. An early example of a system that employed this approach was Tandem’s NonStop system [50].

One shortcoming of dual modular redundancy is that although it can effectively detect single faults; once a fault is detected, the system halts operation and it requires repair. A way to address this limitation is by adding more hardware redundancy to the system, in the form of triple modular redundancy [51]. In triple modular redundancy, three identical microprocessors are used with an additional majority voter. If one of the microprocessors fail, its output is outvoted by the other two microprocessors providing forward system recovery. The system then downgrades into a dual modular redundancy system with the remaining two fault-free microprocessors.

Another traditional fault-tolerance technique used to protect memories, buses, or other microprocessor array structures (e.g., register file) are parity and error correction codes
(ECC) [52]. ECC and parity bits provide a lower overhead solution for data holding hardware structures than modular redundancy. Parity bits are more similar to dual modular redundancy where errors can only be detected but not corrected. On the other hand, ECC resembles triple modular redundancy providing both error detection and forward recovery as the ECC computation masks and corrects the faulty value of a bit. The overhead of parity and ECC bits is relatively low compared to modular redundancy techniques and it comes from the extra storage overhead and the extra logic needed for their computation.

4.2.2 Fault Tolerant Techniques for Modern Processors

In this section, we present several techniques reported in the literature that target fault tolerance in modern processors.

Continuous Execution Checking

DIVA, is an online checker component inserted into the commit stage of a microprocessor pipeline that continuously validates the computation, communication, and control exercised in a complex microprocessor core [53]. The approach unifies all forms of permanent and transient faults, making it capable of detecting computations error due to design bugs, soft errors, and permanent silicon defects. However, a limitation of DIVA is that it does not diagnose the root problem in order to repair the underlying hardware and prevent the errors from occurring again.

An error detection technique for simple processor cores was presented in Argus [54]. The Argus technique continuously checks invariants to detect execution errors. Specifically, Argus, uses run-time invariant checking to detect errors in the control flow, the dataflow, computation, and memory access. The technique provides error detection for errors caused by both permanent silicon defects and transient faults and offers an alternative low-cost fault-tolerance approach when compared to the traditional fault-tolerance approaches.
CHAPTER 4. SOFT ERRORS IN PROCESSOR PIPELINES

Using Resource Redundancy

Hardware redundancy and reconfiguration were used to improve the yield and increase the fault tolerance of future microprocessors in [11]. It was emphasized that inherent resource redundancy, that is abundant in modern microprocessors, should be exploited in both single-core and multi-core processors. Three primary types of inherent redundancy that can potentially be used in a microprocessor were identified: component level redundancy (replicated functional units etc.), array redundancy (spare rows and columns in bit arrays), and dynamic queue redundancy (spare queue entries).

The notion of configurable isolation for low-level fault containment and component reconfiguration through cost-effective modifications to commodity designs was introduced in [55]. Specifically, the proposed mechanism employs dynamic repartitioning of a chip-multiprocessor’s hardware resources into multiple fault zones. Silicon defects are detected at the fault-zone granularity and once a defect is detected, the defective component is disabled and the remaining hardware resources are dynamically repartitioned into new fault zones.

Estimating the Architectural Vulnerability Factor and Fault Masking

The Architectural Vulnerability Factor of a processor structure is defined as the probability that a fault in that structure will result in a visible error in the final output of a program [3]. A bit in which a fault will result in incorrect execution is said to be necessary for architecturally correct execution; these bits are termed ACE bits. All other bits are un-ACE bits. An individual bit may may be ACE for a fraction of the overall execution cycles and un-ACE for the rest. Therefore, the AVF of a single bit can be defined as the fraction of cycles that the bit is ACE. The average AVF of an entire processor can be computed as the weighted average of the AVFs of each structure for systems of reasonable size [56].

In [57], the authors computed runtime program vulnerability to soft errors on four microarchitecture structures (i.e., instruction window, reorder buffer, function units, and wakeup
and concluded that a single performance metric, such as IPC, cache miss, or branch misprediction, is not a good indicator for program vulnerability.

In [58], fault injection was performed on the Illinois Verilog Model. The behavior of the processor for several benchmarks was compared against a golden execution. Fault sites were latches and RAM Arrays. Experiments were performed to determine the level and type of fault masking. It was found that at least 85% of injected single event upsets in the baseline microarchitecture were masked from software.

Detection and Recovery of Pipeline Errors

In [18], a $\pi$ bit was appended with every instruction in order to propagate error information between different parts of the processor. On detecting an error, the instruction queue sets the $\pi$ bit associated with the instruction instead of raising a machine check. At the time of commit, the associated $\pi$ bit is checked to see if the instruction has suffered an error. A machine check is raised in case of an error. The technique is efficient in terms of area, but recovery mechanisms are not discussed.

The errors occurring in the pipeline were classified as Control errors or Component errors in [59]. Error detection for control signals required an additional signal recovery unit. Five different rollback strategies were proposed to recover from errors. Single and multiple bit fault injections were performed on a 64-bit pipeline processor. However, detailed results of the experiments were not presented.

Fault injection experiments for single and multiple bit upsets were performed on a LEON2 processor in [60]. The entire pipeline was triplicated to protect against transient and permanent faults. The pipeline configuration introduced an area overhead of around 26.6 percent and a performance penalty of 23.7 percent in the maximum clock frequency as compared to a non-fault-tolerant processor configuration.

Replicating register values into unused registers to recover from transient faults and soft errors was proposed in [61]. In this approach, if ECC or parity signals an error, the
correct value is taken from the uncorrupted register that holds the copy. In [62] and [63], the authors proposed using hardware-software hybrid schemes which achieve fault tolerance by replicating instructions at the compiler level and using hardware fault detectors that make use of this redundancy. The IBM G5 [64] replicates the frontend and the execution engine, and all instructions are executed twice in parallel. By comparing the output of the instructions, it detects errors. In order to recover from errors, it keeps a copy of the register file. The authors report an increase in total area of 35% due to fault-tolerant techniques.

In [65], it was shown that not all instructions have the same vulnerability. The studies showed that on average, 20% of the instructions are responsible for more than 60% of the total vulnerability. Based on this observation, a microarchitectural technique called Selective Replication was proposed that selectively replicates the most vulnerable instructions in order to detect possible soft errors. The more time an instruction is in a given structure, the more likely it is that it gets a particle strike. If the instruction vulnerability is high, it means that the instruction occupies high number of bits and it spent a long time in the processor components, which makes it more vulnerable. The design selectively reissues and reexecutes those instructions that are above the selected vulnerability threshold. When these selected instructions are placed in the IQ, they are also inserted into the Selective Queue (SQ). Whenever there is an empty port for execution, an instruction in the SQ (whose counterpart in the IQ has already been issued) is issued and executed. Once instructions finish their execution, they keep the result in a widened ROB. When the replica execution finishes, it compares its result against the one stored for validation purposes. In those occasions when the head of the ROB is ready to commit but has not been validated, the commit stalls. The scheme achieved over 60% FIT reduction with less than 4% performance degradation.

Using Multi-threading

Multithreading has been used for error detection and recovery in [66], [67] and [68]. A Simultaneous and Redundantly Threaded (SRT) processor provides transient fault coverage
by running identical copies of the same program simultaneously as independent threads in [68]. The general idea in all the above approaches is to use the multithreading capabilities existing in modern processors to run two copies of the same thread and after execution check the outcome of the instructions to detect the errors and recover from them if possible. This approach causes some degradation in performance. Several works attempted to reduce this performance overhead with better usage of processor resources or using processor idle resources for error checking [69] [70]. In another proposed extension to an out-of-order processor [71], error-detection is achieved by verifying the redundant results of dynamically replicated threads of executions, while the error-recovery scheme employs the instruction-rewind mechanism to restart at a failed instruction. The simulation results of SPEC benchmarks show that in the absence of faults, error detection causes a 2% to 45% reduction in throughput.

Effects of soft errors on microprocessor cores

An analysis regarding the effects of soft errors occurring in both sequential state elements and combinational logic on a DLX microprocessor model was presented in [72]. The error manifestation rates of different architectural blocks were reported. The authors also compared the rates at which soft errors that had already affected architectural state, induce application level failures. It was observed that approximately half of all injected faults have no effect, a quarter result in program crash, and the other quarter result in fail-silent data corruption.

A thorough micro-architectural analysis of the effects of soft errors on a production-level Verilog implementation of an ARM926EJ-S core was carried out in [73]. The authors examined the propagation of faults occurring in both sequential state elements and combinatorial logic. They identified a number of critical differences in error propagation behavior of soft errors occurring in logic gates versus state-based elements.

In [74], function inherent codes are used to detect permanent and transient errors in the
control logic of a high-performance, Alpha-like microprocessor. The RTL description of a decoder was used and extensive simulations were performed to identify potential invariants. However, error correction methods were not discussed.

4.3 Soft Error Detection and Recovery in Inorder Pipelines

In this section, we present an approach to detect transient errors occurring in simple inorder processor pipelines. We also look at possible recovery scenarios that are transparent outside the pipeline (front-end and back-end portions of the processor, operating system, and user code). In other words, pipelines are able to fully recover themselves from transient fault without interaction with the rest of the system or communication beyond Instruction Set Architecture (ISA) boundary. This prevents the effects of the faults from propagating to visible states of the system. In this model, we consider pipeline registers as fault sites since sequential elements are by far more susceptible to SEUs than combinational logic. Nevertheless, we present the extension of this method to handle SEUs in combinational logic as well.

4.3.1 The Proposed Error detection and recovery scheme

As an instruction moves through the pipeline, it gets stored in different stage registers throughout the pipeline along with the associated control signals. If a bit flips due to a particle hit, in one of the stage registers, an incorrect value will propagate through the rest of the pipeline. Our methodology consists of the following three steps:

• Error Detection

• Error Correction

• Error Recovery

We will now explain each of the above steps in detail.
**Error Detection:** The easiest way to detect single bit errors is using parity. Duplication can also be used to detect a single bit error. Error detection can also be tailored for specific stages of the pipeline. For instance, ALU checkers can be used in the Execute stage instead of parity. The Error detection step is required in all the stages in every cycle. Hence, it lies on the critical path. For transient errors occurring in the Combinational logic, any single bit error detection scheme (like parity or duplication), can help us in correcting the error. As soon as we detect an error in combinational logic, we can get the original input values from registers at the input of the stage and recompute the outputs. However, if the stage registers become erroneous, we cannot retrieve the original input values for recomputation as they would be overwritten by the next instruction. So for errors occurring in registers of a stage, we need a scheme which can help us in correcting the errors. Single error correction requires the use of Error Correcting Codes (ECC) as described in [75]. Most modern processors use ECC logic for protecting and correcting bit flips in the cache. ECC requires generation of checksum bits. These bits are then appended to the data bits and passed on to the next stages. The output of the error detection logic in every stage will be an error signal specifying whether an error has been detected or not.

**Error Correction:** This step is needed only for those instructions which are erroneous. Hence, its not in the critical path. When an error signal is raised by the error detection logic, ECC circuitry uses the checksum bits associated with that instruction (data bits) to point to the bit that has flipped. The corrected data is then sent back to the input registers of the stage for re-execution. To reduce the area overhead associated with error correction, it can be implemented as a central unit shared with all stages. As error correction may require several (∼ 10) cycles, a control signal is needed as an output of the error correction logic which is asserted when the correction is complete. Since error happen very infrequently (every billions of cycles), the few cycles for error recovery will not affect the performance (in terms of CPI) at all.

**Error Recovery:** As soon as we detect an error, we recover through the following steps:
• *Stalling the previous stages*: We stall the previous stages so that new instructions do not enter the pipeline while the faulty instruction is being corrected using checksum bits. So, if error is detected in stage $i$, then all previous stages ($i - 1, i - 2$, etc) will be stalled.

• *Flush the output registers of the faulty stage*: In order to prevent the faulty instruction from propagating through the rest of the pipeline, we flush the output registers of the faulty stage. So, if error is detected in stage $i$, a NOOP will pass through stages $i + 1, i + 2$ in subsequent cycles. This means that as long as error correction is not completed, stage $i + 1$ is flushed and NOOP will pass through next stages.

• *Removing the stalls after correction*: When the READY signals coming out of the Error Correction module become one (signifying that correction is complete), we wait one cycle so that the outputs of the faulty stage $i$ are re-computed. We then remove the stalls on the previous stages and the pipeline resumes normally.

Moreover, we need to compute the checksum bits for all the registers on the output of a stage so that it can be used by the next stage. Note that checksum generation cannot be added as an additional stage in the pipeline. This is because we are receiving a new instruction every cycle and we will lose the original (correct) data as it will be overwritten by the new instruction in case error happens. This means that the instruction at which error occurred will be lost and recovery will not be possible. Therefore, checksum generation needs to be done within each stage and this will be in the critical path, i.e. adding delay overhead. The above methodology is shown graphically in Figure 4.3 for one stage.

Figure 4.4 shows a block level representation of the centralized ECC unit. The datapath takes its inputs from a $5:1$ Multiplexer connected with the stage registers. This datapath recomputes the ECC (data and checksum bits) for a bundle of $n + m$ bits (implemented as an XOR network) in order to correct error. The control unit generates control signals for the different stages. These signals include:
Figure 4.3: Proposed error recovery architecture (one stage)

- READY signals: used to inform the stages that the correction is complete. Since ECC packets corrected by the datapath might be smaller than the size of stage register (to reduce its overhead), correcting the entire stage register might take several cycles. Flush/stall signals are active until the correction is completed (READY signals are asserted).

- SELECT signals: sent to the input Multiplexers associated with the stages. They select whether the stage registers take the corrected input or the input from the previous stage. Based on the ERROR signals coming from various stages, the SELECT signals specify which stage is being corrected.

- Flush and Stall signals: used to inform the stages whether they have to flush or stall.

A sequencer synchronizes the steps and ensures that the packets sent to the correction unit are synchronized properly.

The Error Correction and Recovery process may take multiple cycles due to ECC computation. The pipeline needs to be stalled for these cycles and no new instructions will
enter the pipeline. This is similar to what happens during a cache miss. If an operand is not available in the cache, the instructions later in the program code need to be stalled. However, note that particle hits are fairly rare. Even if one incident occurs per millisecond, it translates to one error correction step in a million cycles on modern processors (if the error propagates to a latch). So the performance overhead in terms of CPI is virtually zero. Additionally, since we are using stall and flush (for which modules are already present in modern pipelines), the area overhead will be only for Error Detection unit and ECC. Most modern processors have built-in modules for ECC Checksum Generation and re-computation as data cache is ECC protected. These modules can be reused in the design thus preserving area of the core. However, in case sufficient routing resources are not available, adding an extra ECC unit for the pipelines might be more feasible.

Figure 4.4: Block diagram of the centralized ECC unit

4.3.2 An Example

Consider a five stage MIPS pipeline as shown in Figure 4.5. The clock cycles (CC) are given on the top while instructions are shown in rows. It can be seen that the value of Register \( R_4 \) is forwarded to the next instruction from the outputs of the Execute stage.

Now suppose a bit flip occurs during cycle 4 on the registers in the EX stage. At that point instruction \( I_2 \) was in the Execute stage. It can be seen from Figure 4.6 that this bit flip will not affect instruction \( I_1 \) as it has already reached the MA stage. As soon as the error
is detected, the previous stages (IF and ID) are stalled in cycle 5. Additionally, the execute stage is flushed (NOOP is inserted) so that the erroneous value does not get forwarded to the other parts of the pipeline. In this example it is assumed that ECC computation takes 2 cycles. Hence, in cycles 5 and 6 a NOOP is present in the EX stage. In cycle 7, the outputs of the EX stage are re-evaluated i.e. instruction $I_2$ again gets active in the pipeline. We remove the stalls in cycle 8 and the pipeline continues normally.

### 4.3.3 Errors in Combinational Logic

Transient errors occurring in combinational logic of the pipeline are increasingly contributing to the systems’ failure rate and need to be addressed to ensure reliable operation. Here we discuss possible extensions of the proposed scheme for errors in combinational logic.

**Error Detection:** Different error detection schemes can be used for the different stages of the pipeline, depending on the functionality of each stage.

- **Instruction Fetch:** In the IF stage the combinational logic is generally very simple as no “processing” is done on the fetched instruction. Duplication can be used for this stage and the Error signal will be activated if the result from the two copies is different.

- **Instruction Decode:** The ID stage decodes the instruction and generates control signals for the other stages. Since computing parity is also a combinational circuit, we can add it to the original decode logic as an additional output. The truth table corresponding to the original decode logic is given below:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MA</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD R1,R2,R3</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MA</td>
<td>WB</td>
</tr>
<tr>
<td>SUB R4,R2,R5</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MA</td>
<td>WB</td>
</tr>
<tr>
<td>AND R6,R4,R7</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MA</td>
<td>WB</td>
</tr>
</tbody>
</table>

Figure 4.5: An example of 5-stage MIPS pipeline with data forwarding from ALU
CHAPTER 4. SOFT ERRORS IN PROCESSOR PIPELINES

Figure 4.6: Example of error recovery: a bit flip occurs in clock cycle 4
to the additional output is set such that for each input combination, the outputs
(including the additional one) form even (odd) parity.

- Execute: Since ALU performs arithmetic and logical functions, we can use Residue
  Codes for error detection. In these codes, we attach a check symbol “\(C(X) = X \mod A\)”
to each operand X. ‘A’ is called the check modulus. The codes are pre-
  served under a set of arithmetic operations. Several well known residue codes are
described in [76].

- Memory Access and Write Back: Since the combinational logic associated with these
  stages is typically very small, duplication can be used for error detection.

4.3.4 Implementation and Results
We implemented the proposed error detection and recovery scheme on a 32-bit MIPS based
processor described in Verilog at the RT-Level. The processor employs a five stage pipeline
and is capable of hazard detection and data forwarding. Every stage has registers on its
inputs and outputs. We use 32-bit combinational ECC modules for generating checksum
bits in every stage. Instead of having a large XOR tree for checksum generation of multiple 32-bit registers, we used several 32-bit checksum generation modules in every stage. Each module operates on one 32-bit register and generates a 7-bit checksum. For error recovery, we used the checksum bits to correct the error. A 32-bit ECC module receives 39 (32 data and 7 checksum) bits at a time and corrects the error. This operation may take multiple cycles. However, since it is not on the critical path (recovery is done only when an error flag is raised by a stage in every billions of cycles), it does not affect the performance at all. Note that transient errors are not very frequent and the impact on performance would be minimal.

The processor was synthesized using Synopsys Tools. We used the OSU 0.25 micron library for synthesis. Level of mapping effort was Medium. In Table 4.1 the area and timing information of the original and modified architecture are presented. The original module does not contain any error detection or correction. For each stage, the modified architecture instantiates eight 32-bit checksum generators to produce checksum bits for the next stage. Each stage also includes a parity based error detection unit. The 32-bit ECC correction unit takes 19.22 units (i.e., one clock cycle) to complete. If one unit is used, correcting eight 32-bit registers takes 8 cycles. Note that the correction can be done in parallel. So if we have two units in parallel, eight 32-bit registers will require 4 cycles for correction. It can be seen that the technique introduces very little area overhead and impact on performance is minimal.

We have also verified the correctness of the proposed approach by performing extensive fault injections in the Register-Transfer (RT) model described in Verilog. The simulations were carried out using Modelsim. In all cases, the modified architecture was able to successfully recover from all injected errors with correct results.

Our results show that full recovery is achievable with very low area (14.9%) and delay (25.8%) overhead. Note that, the schemes presented above will not be applicable to out-of-order pipelines. Such pipelines are covered in the following section.
Table 4.1: Area and timing (maximum clock period) of the Original and Modified MIPS architectures

<table>
<thead>
<tr>
<th></th>
<th>Area</th>
<th>Timing (clock period)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original MIPS</td>
<td>74,928</td>
<td>16.28</td>
</tr>
<tr>
<td>Modified MIPS</td>
<td>86,135</td>
<td>20.49</td>
</tr>
<tr>
<td>Overhead</td>
<td>14.9%</td>
<td>25.8%</td>
</tr>
</tbody>
</table>

4.4 Error Detection and Recovery in Superscalar Pipelines

As mentioned in [45] [77], most multi-core systems are built on simple cores, i.e. those having simple pipelines. However, processors used in high performance and high availability systems frequently employ deep pipelines and have the ability to issue multiple instructions per cycle [21] [46] [78]. The instructions are executed in parallel in separate functional units whenever possible. This form of Instruction Level Parallelism (ILP) is called Superscalar execution.

Traditionally, uncertainties due to process, voltage and temperature variations are addressed by considering design margins and operating the die at conservative voltage and frequency points such that sufficient safety margins exist [79]. As process geometries shrink, the unacceptable performance and power impacts of pessimistic design margining has led to an increased interest in adaptive online techniques [80] [81]. Reducing design margins helps in reducing cost / power ratio and increasing performance [79] [80]. The additional errors encountered as a result of reducing the design margins and voltage scaling can be detected and corrected by online techniques. It is important to note, that errors occurring due to aggressive voltage, power and design margin scalings are by far more frequent than radiation-induced (and other transient) errors. Hence, fast and effective recovery schemes are extremely crucial to get the benefit of such approaches.

As mentioned in Section 4.2, several techniques have been proposed in the literature
which look at detection and/or recovery of transient errors in superscalar pipelines. Although the techniques look at multiple ways to detect transient errors, most of these techniques either propose flushing the pipeline for error recovery \[53\] \[82\] \[54\] \[66\] \[67\], rely on ECC \[83\] or triplication \[60\]. While ECC and triplication introduce significant area overhead, flushing makes the techniques inapplicable for reducing design margins \[79\] \[80\] due to high recovery latency. Moreover, flushing in deep pipelines requires checkpointing mechanisms which introduce significant overheads in the design.

In this section, we propose schemes for protecting the front-end logic of present-day superscalar processors. The front-end logic is comprised of several structures (like Rename Table, Issue Queue and Re-order buffers) which contain critical information about instructions moving through the pipeline. As the instructions reside in these structures for multiple cycles, the Architecture Vulnerability Factor (AVF) of these structures is significantly high \[84\]. The proposed schemes look at the lifetime of instructions in various pipeline structures, utilize the inherent redundancy present in the front-end logic of the pipeline and increase it to a minimal level, in order to detect and recover from transient errors.

While the above mentioned techniques look at \emph{either} the hardware implementation \[53\] \[82\] \[83\] or a cycle-accurate architecture simulator \[66\] \[45\] \[70\], we implement our error detection and recovery techniques on \emph{both} Register-Transfer Level (RTL) and a cycle-accurate performance simulator. This allows us to analyze various implications of our proposed modifications and obtain accurate area overheads and critical path delay (from the RTL). It and allows us to run realistic workloads on the cycle-accurate simulator to get performance (CPI) and power estimates.

Our proposed schemes provide an efficient recovery (low latency) and are transparent to the user and operating system. Moreover, they do not require checkpointing which is expensive in area and has significant overhead.

In the following sections, we present possible error detection and recovery schemes for
transient errors occurring inside the ROB, IQ and Rename Table. The schemes take advantage of the information redundancy already present in these structures. We compare different tags associated with the instruction at the time of issue and commit. In case the tags do not match, we stall the pipeline and attempt to recover the information from other structures before the instructions are issued and committed. As the instructions remain in ROB for a longer time compared to IQ, the techniques mentioned below require the instructions to be kept in the IQ until they commit.

4.4.1 Error Detection and Recovery in ROB

In this section, we present possible error detection and recovery schemes for transient errors occurring inside the ROB. The errors may occur in the head, tail pointers of the circular buffer or in various fields of the stored instructions. As demonstrated in Figure 4.7, the lifetime of an instruction in ROB is greater than that in the Issue Queue. Hence, instructions are vulnerable for a longer time in ROB compared to IQ.

![Figure 4.7: Lifetime of an instruction in IQ, ROB and Rename table](image)

It is important to note that any bit-flips in the instructions will cause latent errors. These will be realized when an instruction reaches the head of the ROB. Due to this reason, we are only interested in the contents of an instruction when it reaches the head of the ROB. This is the time when it tries to commit. Each ROB entry contains the ROB ID (Robid), Architecture Register tag, Physical Register tag and Flags.
To enable error detection, we add certain signals and components to the front-end logic. In doing so, we try to minimize the number of extra ports needed in order to transfer the information between different structures. The instruction at the Head of the ROB is referred to as $\text{Inst}_{\text{rob}_\text{out}}$. At the time of commit, we compare various fields of this instruction with the corresponding fields of the same instruction in IQ. In case of a mis-match, an error ($\text{Error}_{\text{commit}}$) is flagged. This results in stalling the pipeline until the error is corrected. The error detection logic for the ROB is shown in Fig. 4.8(a).

Figure 4.8: Error Detection at (a) Commit time (b) Issue time
Fields and structures added for Redundancy.

In order to provide minimal redundancy needed for proper error detection and correction, we added certain fields in ROB. The shaded fields in Fig. 4.8(a) show those that have been added to all ROB entries. We also added the following resources inside the ROB, in order to detect errors in the Head and Tail pointers of the ROB:

- **regHT**: The regHT register has a width equal to size of the ROB. Each bit in this register represents whether the corresponding ROB entry is at the head or tail of ROB or not. So in error free conditions, if ROBid of the Head instruction is $x$ and that of the tail instruction is $y$, $\text{regHT}[x] = 1$ and $\text{regHT}[y] = 1$. All other bits of $\text{regHT}$ will be 0 as shown in Fig. 4.8(a).

- **Length**: A register, length needs to be added to the structure of the ROB. Its value will be computed by using:
  \[
  \text{length} = |\text{Head} - \text{Tail}|
  \]

Error Detection Logic.

We also add some signals (shown in Red), that have been added for error detection and correction. Three signals ($\text{ROB\_arch}, \text{ROB\_phys}$ and $\text{ROB\_type}$) move from ROB to IQ carrying another copy of these fields in case an error occurs in IQ. Additional signals go from IQ to ROB corresponding to the architectural register field ($\text{IQ\_arch}$) and physical register field ($\text{IQ\_phys}$) of the instruction at the Head of ROB. An additional 2-bit signal ($\text{ER\_Type}$) also comes from IQ into the ROB which is defined as follows:

Bit 0: The Execute flag of the instruction showing that the instruction has already executed.

Bit 1: The Source Registers have an error or not. This will be explained in the following section.

The register regHT is updated every time an instruction enters or leaves the ROB. Note that ROBid of such instruction is known using $\text{Inst\_rob\_in.Robid}$ and $\text{Inst\_rob\_out.Robid}$.
The update procedure is shown in Fig. 4.9.

![Figure 4.9: Updating the regHT register](image)

The bits in red are those that need to be changed. We update the \( \text{regHT} \) register as follows. Suppose \( t \) is the Robid assigned to the entering instruction and \( h \) is the Robid of the instruction that just committed.

*When an instruction enters the ROB:

\[
\text{regHT}[t] \leftarrow 1 \quad \text{and} \quad \text{regHT}[t - 1] \leftarrow 0
\]

*When an instruction commits:

\[
\text{regHT}[h] \leftarrow 0 \quad \text{and} \quad \text{regHT}[h + 1] \leftarrow 1
\]

Note that the operations done above are modulo \( \text{ROB\_Size} \). Also, we applied the condition that if \( t \) is 0, \( t - 1 \) will be treated as \( \text{ROB\_Size} - 1 \).

An input signal \( \text{Correct} \) is added to IQ and ROB. This signifies that a correction is needed in that module. Once the correction is complete, \( \text{ROB}_{\text{ok}} \) and \( \text{IQ}_{\text{ok}} \) are issued meaning that the error has been corrected and stall can be removed.

**Error Detection Methodology.**

As shown in Fig. 4.8(a), the \( \text{(Robid)} \) of the Head instruction is sent to the IQ. In response, the IQ transmits the Architecture and Physical Register tags \( \text{IQ\_arch} \) and \( \text{IQ\_phys} \) of the corresponding instruction. These tags are compared to the tags of the Head instruction \( \text{(Inst\_rob\_out.Arch} \) and \( \text{Inst\_rob\_out.Phys} \). In case of a mis-match, an error \( \text{(Error\_commit)} \) is flagged. This results in stalling the pipeline until the error is corrected. The following comparisons should hold regarding architectural and physical registers of the instruction at head of ROB:
\(IQ_{\text{arch}} == \text{Inst}_{\text{rob.out}}.\text{Arch}\) and,
\(IQ_{\text{phys}} == \text{Inst}_{\text{rob.out}}.\text{Phys}\).

Another important field in a ROB entry is the \textit{Ready} flag. When an instruction finishes execution and is ready to commit, this flag becomes 1. In order to detect a bit flip in this flag, the field \(\text{Inst}_{\text{rob.out}}.\text{Flags.Ready}\) is compared with \(\text{bit 0 of } ER_{\text{type}}\) from IQ. Hence, in case of no errors,

\[ER_{\text{type}}[0] == \text{Inst}_{\text{rob.out}}.\text{Flags.Ready}\]

In case of a mismatch, the instruction is re-issued from IQ.

For detecting errors involving head and tail pointers, we check the \textit{length} of the ROB. The \textit{length} register can keep track of the number of instructions in the ROB and to help us in error detection. This register will need updating, whenever an instruction enters or retires from the ROB. In case \(\text{length} \neq |\text{Head} - \text{Tail}|\) an error can be flagged. The comparisons needed are shown in Figure 4.10.

![Figure 4.10: Logic for Error Detection in Head and Tail pointers](image)

\textbf{Error Correction and Recovery in ROB}

A high on the \textit{Correct} signal will mean that correction needs to be done. Since the signal is shared between IQ and ROB, we check the \textit{Error\_commit} bit in ROB. If this is 1, then correction needs to be done at the Head of the ROB. ROB updates its Head entry by getting the correct values from IQ as shown below:

\[
\text{if } (\text{Correct} == 1 \text{ AND } \text{Error\_commit} == 1)
\]

\[\text{Inst}_{\text{rob.out}}.\text{Arch} \leftarrow IQ_{\text{arch}}\]
\[ \text{Inst}_{\text{rob}} \cdot \text{out} \cdot \text{Phys} \leftarrow \text{IQ}_{\text{phys}} \]
\[ \text{ROB}_{\text{ok}} \leftarrow 1 \]

As shown above in case of an error in either the Architecture register tag or the Physical register tag, we copy both the tags from IQ. Additionally, as mentioned above the \( \text{ER}_{\text{Type}[1]} \) signal coming in to the ROB (from IQ), signifies that there is an error in Source operands of an IQ entry corresponding to the \( \text{IQ}_{\text{Robid}} \) input. In this case, the correct Source registers tags are sent to IQ as follows:

\[ \text{ROB}_{\text{arch}} \leftarrow \text{ROB}[\text{Robid}].\text{SrcA} \text{ and } \text{ROB}_{\text{phys}} \leftarrow \text{ROB}[\text{Robid}].\text{SrcB} \]

If an error occurs in the Head or Tail pointer, we monitor the Head and tail bits of \( \text{regHT} \) as well as the \( \text{length} \) register. We can recover as follows:

- If \((\text{regHT}[\text{Head}] = 0) \text{ AND } (\text{regHT}[\text{Tail}] = 1) \text{ AND } (\text{length} \neq |\text{Head} - \text{Tail}|)\):
  \( \text{Error is in the Head Pointer.} \)
  \( \text{Head} \leftarrow w \)
  where \( w \) is the bit in \( \text{regHT} \) (other than the tail) which is 1.

- If \((\text{regHT}[\text{Head}] = 1) \text{ AND } (\text{regHT}[\text{Tail}] = 0) \text{ AND } (\text{length} \neq |\text{Head} - \text{Tail}|)\):
  \( \text{Error is in the Tail Pointer.} \)
  \( \text{Tail} \leftarrow w \)
  where \( w \) is the bit in \( \text{regHT} \) (other than the head) which is 1.

- If \((\text{regHT}[\text{Head}] = 1) \text{ AND } (\text{regHT}[\text{Tail}] = 1) \text{ AND } (\text{length} \neq |\text{Head} - \text{Tail}|)\):
  \( \text{Error is in length register. We recompute it to recover.} \)
  \( \text{length} \leftarrow |\text{Head} - \text{Tail}| \)

- If \((\text{regHT}[\text{Head}] = 0) \text{ AND } (\text{regHT}[\text{Tail}] = 1) \text{ AND } (\text{length} = |\text{Head} - \text{Tail}|)\):
  \( \text{regHT}[\text{Head}] \) is incorrect.
  \( \text{regHT}[\text{Head}] \leftarrow 1 \)
• If \((\text{regHT}[\text{Head}] = 1) \text{ AND } (\text{regHT}[\text{Tail}] = 0) \text{ AND } (\text{length} = |\text{Head} - \text{Tail}|)\):
  \(\text{regHT}[\text{Tail}]\) is incorrect.
  \(\text{regHT}[\text{Tail}] \leftarrow 1\)

Note that, any single bit error occurring in Head, Tail, Length or \(\text{regHT}\) can be detected and corrected.

### 4.4.2 Error Detection and Recovery in IQ

The **critical time** of an instruction in IQ is from the time it is entered in IQ by the rename unit, until it is ready to execute, i.e. issue time. During critical time, the instruction is vulnerable to errors. Therefore, the detection (as well as correction and recovery) has to be done at the end of critical time period. This ensures that no further errors in IQ will affect that instruction. As noted above, the instructions need to be kept in the IQ until they commit, in order to provide minimal redundancy required for comparisons.

**Fields and structures added for Redundancy.** The shaded fields in Fig. 4.8(b) represent those fields that have been added to the baseline architecture in order to provide minimal redundancy for error detection and correction. These include the Architectural register field and a field \(\text{SrcP}\) which is the result of XORing the two Source registers as described later.

**Error Detection Logic.** The error detection logic for IQ is shown in Fig. 4.8(b). \(\text{Inst}_{\text{iq} \_\text{out}}\) is the instruction that is about to be issued (its source operands are ready and execution unit needed is available). The Error detection logic is similar to that of the ROB. The \(\text{Robid}\) of the instruction to be issued is sent to the ROB (using \(\text{IQ}_{\_}\text{Robid}\) signal), that in turn, sends the corresponding architectural and physical register tags (\(\text{ROB}_{\_}\text{arch}\) and \(\text{ROB}_{\_}\text{phys}\)) and the Type field (\(\text{ROB}_{\_}\text{type}\)).
Error Detection Methodology.

The error detection involves comparing the physical register values of $Inst_{iq\_out}$ with the value of the same field in the corresponding instruction in ROB. If there is a mis-match, $Error\_issue$ is flagged.

For the $Inst_{iq\_out}\_Type$ field, we use a one-hot encoding as the number of different types of execution units is very limited. In case more than one bit is 1, the correct value is copied from $ROB\_type$. For the source registers, we propose a parity based (rather than a comparison based) scheme for detecting errors as this information is not present in all structures. The scheme is described below.

The logic of the Rename stage is modified such that it generates a value $SrcP$ by XORing $SrcA$ and $SrcB$ fields of the instruction that is being sent to IQ. At the time of issue, this parity is re-computed and output signal $ER\_type$ is updated as follows:

$$regSrcP2 \leftarrow SrcA \oplus SrcB$$

if $regSrcP2 \neq SrcP$

$$ER\_type[1] \leftarrow 1$$

**Error Correction and Recovery Scheme for IQ**

In order to have proper recovery, the Rename (and previous stages) need to be stalled so that new entries are not added to ROB, IQ and Rename table. Moreover, the error correction logic is not on the critical path and is activated only if $Error\_commit$ or $Error\_issue$ is asserted. The inputs to the circuit in Figure 4.11 consist of the error signals generated.
by error detection circuitry of ROB (Error_{commit}) and IQ (Error_{issue}). We add a 2 input multiplexer that gets its data inputs from the architectural registers tags of $\text{Inst}_{\text{rob.out}}$ and $\text{Inst}_{\text{iq.out}}$ and its control input based on whether we need the physical register corresponding to the desired IQ entry or the Head of the ROB. Another 2-input multiplexer, selects between $\text{Inst}_{\text{rob.out.phys}}$ and $\text{Inst}_{\text{iq.out.phys}}$. A 2:4 decoder generates the appropriate signals for the select logic of the multiplexers. The output 00 of the decoder goes to the EN input of the two multiplexers. Hence, in case there is no error, both the multiplexers are turned off. Otherwise, the appropriate Physical register tag is compared with $\text{Ren}_{\text{phys.out}}$, which is the physical register value obtained from the Rename Table. In case of a mis-match, a signal $\text{Correct}$ is flagged that is sent to both IQ and ROB.

If the $\text{Correct}$ signal is high, we check the Error_{issue} bit in IQ. If this is 1, then correction needs to be done in the $\text{Inst}_{\text{iq.out}}$ instruction. The Robid field of this instruction is sent to the ROB, and the correct values for Physical register tag and Type are obtained. The entry is updated as follows:

\[
\text{if (Correct == 1 AND Error_{issue} == 1)} \\
\text{\quad Inst}_{\text{iq.out.Phys}} \leftarrow \text{ROB.phys} \\
\text{\quad Inst}_{\text{iq.out.Type}} \leftarrow \text{ROB.Type} \\
\text{\quad IQ}_{\text{ok}} \leftarrow 1}
\]

In this case, we do not need the Architectural register tag as it is not required for execution.

If $\text{ER.Type}[1]$ is equal to 1 at issue time, there is an error in $\text{Inst}_{\text{iq.out.SrcA}}$ or $\text{Inst}_{\text{iq.out.SrcB}}$. In this case, we obtain the correct values of the Source registers from the ROB as mentioned above. The signals carrying two Source Register tags were multiplexed with those carrying Architecture and Physical Registers to reduce the number of ports.

It should be noted that our comparison-based schemes described above will work only if correct assignments for source and destination tags were made in the Rename stage. If the Rename logic has errors such as providing the same destination register to two instructions
or assigning wrong physical registers as source operands, our comparisons and subsequent execution of the instructions will become invalid. Hence, to ensure proper execution, the Rename logic needs to be protected using mechanisms such as those presented in [85].

4.4.3 Handling multiple commits

Superscalar processors have the ability to commit multiple instructions in one cycle. This is generally handled at Dispatch time. So instructions that are not dependent on each other (and do not disturb the program order) are dispatched simultaneously to IQ and ROB. In our case, a maximum of 4 instructions were allowed to commit simultaneously. Hence, in effect, the Head pointer was pointing to 4 instructions. In IVM, this is implemented as 4 separate ROBs with the same Head and Tail pointers. All our error detection mechanisms are based on comparing the value of various fields of an instruction in ROB to the corresponding fields of the same instruction in IQ. Whenever more than one instruction needed to commit in one cycle, we queued the Robid of the instructions and sent them to the IQ one after the other. The additional number of cycles taken for 2, 3 or 4 simultaneous commits were noted and added to the cycle count in M5, whenever multiple instructions were committed at the same time in M5. Note that, this was done because the CPI in the non-modified circuit for the benchmarks that we studied was in almost all cases more than 1. Hence, the workload was not sufficiently parallel in order to allow multiple commits. If however, extensive parallelism exists, the above approach for handling multiple commits might result in increasing the CPI considerably. One alternative in this case is to have multiple units for comparison. This will improve the CPI but would be accompanied by an increase in the area of the core.

4.4.4 Implementation issues

We implemented the techniques described in Sections 4.4.1 and 4.4.2 on top of an RT-Level implementation of Alpha processor. A synthesizable subset of the open-source Illinois
Verilog Model (IVM) was used [86]. In order to get real performance overheads (cycle counts), we also implemented our schemes on top of a cycle-accurate performance simulator, M5 [49]. It can simulate an out-of-order, superscalar, pipelined, simultaneous multithreading (SMT) model of an Alpha processor.

The parameters used for various structures in the pipeline are mentioned in Table 4.2.

<table>
<thead>
<tr>
<th>Configuration</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 Cache</td>
<td>64 KB, 2-way set associative, Blk Size 64</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>2 MB, 8-way set associative, Blk Size 64</td>
</tr>
<tr>
<td>Functional Units</td>
<td></td>
</tr>
<tr>
<td>Int ALU</td>
<td>6</td>
</tr>
<tr>
<td>Int Mult</td>
<td>2</td>
</tr>
<tr>
<td>FP Adder</td>
<td>4</td>
</tr>
<tr>
<td>FP Mult</td>
<td>2</td>
</tr>
<tr>
<td>FP Div</td>
<td>2</td>
</tr>
<tr>
<td>Pipeline Parameters</td>
<td></td>
</tr>
<tr>
<td>Load Queue</td>
<td>32 entries</td>
</tr>
<tr>
<td>Store Queue</td>
<td>32 entries</td>
</tr>
<tr>
<td>Pipeline Width</td>
<td>4</td>
</tr>
<tr>
<td>Issue Queue</td>
<td>32 entries</td>
</tr>
<tr>
<td>ROB</td>
<td>64 entries</td>
</tr>
<tr>
<td>Registers</td>
<td>128</td>
</tr>
</tbody>
</table>

Although area overhead of our schemes could be 'estimated' based on the additional fields needed in the pipeline structures, such estimates hide the real hardware costs in terms of extra ports added to the design, area of comparators etc. Hence, in order to obtain accurate area estimates, we used an RTL-based implementation.

Below we itemize some challenges and issues faced while implementing the RT-Level code:

**Number of ports**: In order to minimize the increase in number of I/O ports in the design, we have used control signals such as ER.Type to signal errors and multiplexed various type
of tags on the same lines. For instance, in order to transfer the tags of \textit{SrcA} and \textit{SrcB} from ROB, we use the same lines on which \textit{Arch} and \textit{Phys} were transferred instead of having separate ports for \textit{SrcA} and \textit{SrcB}.

**Comparisons:** The comparators needed for the implementation are at most 7 bits in width. Instead of comparing all the fields simultaneously (which would require a large comparator), we compare one field at a time. Since the comparisons are now quick, the cycle time is affected minimally.

\textit{regHT Register}: One alternative of implementing this was to add a HT bit with every entry of ROB and update it whenever the entry becomes a Head or Tail. This approach however results in 4 writes to the ROB in every cycle. To overcome this, we implemented \textit{regHT} as a register inside the ROB (but outside the CAM). Since Head and Tail are internal registers to the ROB, we did not implement \textit{regHT} at the higher level module as this would need additional read/write ports. Although, adding a register inside the ROB would increase susceptibility to errors, we put mechanisms to be able to detect errors in this register as shown in Sections 4.4.1 and 4.4.2.

**Handling multiple commits**

Superscalar processors have the ability to commit multiple instructions in one cycle. This is generally handled at Dispatch time. So instructions that are not dependent on each other (and do not disturb the program order) are dispatched simultaneously to IQ and ROB. In our case, a maximum of 4 instructions were allowed to dispatch/commit simultaneously. Hence, in effect, the Head pointer was pointing to 4 instructions. In IVM, this is implemented as 4 separate ROBs with the same Head and Tail pointers. All our error detection mechanisms are based on comparing the value of various fields of an instruction in ROB to the corresponding fields of the same instruction in IQ. Whenever more than one instruction needed to commit in one cycle, we queued the Robid of the instructions and sent them to the IQ (for comparison) one after the other. Each simultaneous commit took an additional
cycle. So 2 commits took 1 extra cycle, 3 commits took 2 extra cycles and so on. The additional number of cycles taken for 2, 3 or 4 simultaneous commits were added to the cycle count in M5, whenever multiple instructions were committed at the same time in M5. Note that, this was done because the CPI in the non-modified circuit for the benchmarks that we studied was in almost all cases more than 1. Hence, the workload was not sufficiently parallel in order to allow multiple commits. If however, extensive parallelism exists, the above approach for handling multiple commits might result in increasing the CPI considerably. One alternative in this case is to have multiple units for comparison. This will improve the CPI but would be accompanied by an increase in the area of the core. Multiple issues were not a problem in this case as we are only doing comparisons when the Ready_to_issue flag becomes 1 (and not at dispatch time). This flag is asserted when the operands are ready and dependencies are resolved. So the only effect is that instructions are issued slightly later because of comparisons. Hence, this delay is already accounted for in the results.

In order to get reasonable power estimates we use a power simulator McPat \cite{87}. McPAT is an integrated power and area modeling framework that supports comprehensive design space exploration for multicore and manycore processor configurations ranging from 90nm to 22nm and beyond. At the microarchitectural level, McPAT includes models for the fundamental components of a chip multiprocessor, including in-order and out-of-order processor cores, networks-on-chip, shared caches, integrated memory controllers, and multiple-domain clocking. McPAT runs separately from M5 and only reads performance statistics from it therefore, its impact on the simulation speed of M5 is minimal.

4.4.5 Results and Discussion

In this section, we present detailed results of our fault injection experiments. The overhead in area, delay, performance and power is reported using RT-level Verilog and M5 performance simulator.
CHAPTER 4. SOFT ERRORS IN PROCESSOR PIPELINES

Validation using RTL & Performance simulator

The fault sites were registers holding the instructions in the ROB and IQ. Additional registers like length and htreg as well as Head and Tail pointers were also marked as fault sites. In the RT-Level code, bit flips were injected under single fault assumptions and the detailed values at every clock cycle were observed using the Synopsys VCS Simulator. All the bit flips were successfully detected and corrected by our proposed approach. In M5 performance simulator, initially, 1 million instructions were allowed to execute to warm up the core. Then after a random number of instructions were committed, the desired field (e.g. Destination Register in the ROB) was XORed randomly with $2^i$, to inject bit flip in the $i^{th}$ position. The experiment was repeated 10 times for each fault site. The output of each experiment was compared with that of a golden run (with no fault injection). In all cases the outputs matched exactly.

The proposed error detection and recovery schemes was verified by executing SPEC2000 benchmarks on 3 implementations of Alpha architecture using M5. One was the original implementation obtained from [88]. The second implementation included error detection and correction logic without injecting errors (to compute the performance overhead of error detection scheme). The third implementation included error detection and recovery capabilities with fault injection (to compute the recovery latency of error correction). All injected errors were correctly detected and successfully recovered.

Performance overhead using M5 simulator

The performance impact of running SPEC benchmarks on original and proposed architectures can be seen in Table 4.3.

In Table 4.3 Columns 2 and 3 contain the performance numbers in Cycles per Instruction (CPI) for original and modified architecture, respectively. The overhead for each benchmark, in terms of CPI, is shown in Column 4. Column 5 shows the recovery latency (i.e. time to correct and repair) in cycles for SPEC2000 benchmarks. It can be seen from
Table 4.3: Performance overhead of error detection and correction for SPEC2000 benchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Original (CPI)</th>
<th>Modified (CPI)</th>
<th>Overhead %</th>
<th>Recovery Latency (Cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td>art</td>
<td>1.218</td>
<td>1.321</td>
<td>8.21</td>
<td>17</td>
</tr>
<tr>
<td>bzip</td>
<td>1.196</td>
<td>1.274</td>
<td>6.52</td>
<td>15</td>
</tr>
<tr>
<td>gcc</td>
<td>1.361</td>
<td>1.483</td>
<td>8.96</td>
<td>11</td>
</tr>
<tr>
<td>gzip</td>
<td>1.314</td>
<td>1.466</td>
<td>11.56</td>
<td>13</td>
</tr>
<tr>
<td>mcf</td>
<td>1.268</td>
<td>1.364</td>
<td>7.57</td>
<td>12</td>
</tr>
<tr>
<td>mesa</td>
<td>1.188</td>
<td>1.269</td>
<td>6.81</td>
<td>13</td>
</tr>
<tr>
<td>crafty</td>
<td>1.084</td>
<td>1.179</td>
<td>8.76</td>
<td>11</td>
</tr>
<tr>
<td>parser</td>
<td>1.117</td>
<td>1.192</td>
<td>6.71</td>
<td>9</td>
</tr>
<tr>
<td>twolf</td>
<td>1.291</td>
<td>1.388</td>
<td>7.51</td>
<td>21</td>
</tr>
<tr>
<td>swim</td>
<td>1.286</td>
<td>1.394</td>
<td>8.39</td>
<td>18</td>
</tr>
<tr>
<td>applu</td>
<td>1.336</td>
<td>1.447</td>
<td>8.30</td>
<td>18</td>
</tr>
<tr>
<td>ammp</td>
<td>1.241</td>
<td>1.331</td>
<td>7.25</td>
<td>12</td>
</tr>
<tr>
<td>equake</td>
<td>1.511</td>
<td>1.609</td>
<td>6.48</td>
<td>16</td>
</tr>
<tr>
<td>galgel</td>
<td>1.122</td>
<td>1.218</td>
<td>8.56</td>
<td>15</td>
</tr>
<tr>
<td>lucas</td>
<td>1.314</td>
<td>1.422</td>
<td>8.21</td>
<td>19</td>
</tr>
<tr>
<td>Average</td>
<td></td>
<td></td>
<td>7.99</td>
<td>14.66</td>
</tr>
</tbody>
</table>

Table 4.3 shows that the overhead in cycles per instruction for SPEC2000 benchmarks is less than 8% on average. The increase in CPI can be attributed to the requirement that instructions remain in the IQ until they commit. This puts extra pressure on IQ and some instructions might need to be stalled until there is an available slot in the IQ.

Error correction and recovery involves, stalling the fetching of new instructions, correcting the bit and resuming the pipeline. Note that since we are not re-fetching the faulty instructions, the recovery overhead is low (15 cycles on average).

**Area overhead using RTL**

We add different fields in ROB and IQ in order to provide minimal redundancy. Moreover, error detection circuitry (which mainly consists of 8-bit comparators) as well as some logic for error correction needs to be added to the core. In order to obtain an accurate estimate of increase in area, the RT-Level code was compiled using the Synopsys Design Compiler with
Generic Standard Cell (GSC) 180 nm libraries. The cell area for various structures obtained after synthesis is shown in Table 4.4. The area overhead for ROB is around 5% which is due to the fact that we add additional fields in ROB entries (for providing redundancy) as well as the register $htreg$ and the logic for updating it. The area overhead for Issue Queue is much smaller as we only add one additional field per entry and some logic for generating the XOR of source operands. The logic for comparing the tags as well as the control signals for error correction is outside ROB and IQ. The area of ‘core’ mentioned in Table 4.4 is that of a synthesizable subset of the core that does not contain components like Instruction cache, RAM etc. It can be seen that the area overhead for the core is around 0.5%.

### Delay overhead using RTL

Using Synopsys tools, we obtained the critical path delay, i.e. the maximum possible frequency at which the processor can operate. Since we are adding error detection circuitry to the critical path, the maximum possible operating frequency is reduced. The maximum frequency for the original core was 640 MHz. This reduced to 618 MHz after we added the detection and correction circuitry. Hence the degradation in clock rate was 3.4%.

### Power overhead using McPat power simulator

The McPAT tool was used to obtain peak and average power estimates based on the statistics generated by M5. The tool required configuration values of various structures like size and structure of cache, Issue Queue, ROB etc. Additionally, statistics generated by the M5 simulator were also passed on to McPat using a couple of Perl scripts. Figure 4.12

<table>
<thead>
<tr>
<th>Resource</th>
<th>Original cell area</th>
<th>Modified cell area</th>
<th>Overhead %</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROB</td>
<td>674,217</td>
<td>710,020</td>
<td>5.31</td>
</tr>
<tr>
<td>IQ</td>
<td>891,009</td>
<td>902,150</td>
<td>1.25</td>
</tr>
<tr>
<td>core</td>
<td>8,937,445</td>
<td>8,984,560</td>
<td>0.53</td>
</tr>
</tbody>
</table>
CHAPTER 4. SOFT ERRORS IN PROCESSOR PIPELINES

graphically shows the overhead in average and peak power for various benchmarks.

![Power Consumption Graph]

Figure 4.12: Effect on Peak and Average Power for SPEC2000 Benchmarks

As can be seen from Figure 4.12, our proposed technique causes an increase of 1% in Peak power and less than 2% in Average power. Note that the power estimates shown above are for the entire core. The overhead is due to the fact that we are not increasing the size of power hungry resources like cache, register file etc. The power consumed by ROB and IQ are typically 3 - 4% of the power of overall core. Hence, small increases in their area would not lead to huge increase in power consumption of the core as can be seen from the results.

**Effect of increase in size of IQ and ROB**

By increasing the size of IQ to 64 (same as ROB), the performance (CPI) overhead was reduced to only 4.1% on average (≈ 4% less) as shown in Figure 4.13. This was because more IQ slots were available and instructions did not need to stall additionally for an IQ slot. This improvement in performance was however, accompanied by a 3.9% area overhead (≈ 3.4% more). This is in comparison to the original core with 32 IQ entries.

The overhead for average power was 2.8% (≈ 0.8% more), compared to a non-modified core with an IQ size of 64. When compared with the original core (IQ size 32), the average power overhead was 6.4%. The results are shown graphically in Figure 4.14. In applications where area and power are secondary to performance, such an approach might be feasible.
CHAPTER 4. SOFT ERRORS IN PROCESSOR PIPELINES

Figure 4.13: Effect of changing IQ size on Performance

Figure 4.14: Effect of changing IQ size on Average Power
Handling multiple bit errors

Multiple bit errors in the Source register fields in IQ can only be detected, as the detection strategy is based on parity. In other fields, since our approach is based on comparisons, multiple bit errors can be detected and corrected as long as they do not corrupt the same fields of an instruction in both ROB and IQ. If however, errors affect the same field of an instruction in two or more structures, the proposed techniques can only detect such multiple bit errors. Then, typical error recovery, involving flushing the pipeline, restoring the state from a checkpoint and re-executing the instructions, can be performed.

4.5 Summary

In this chapter, we presented several online techniques for error detection, correction and recovery in processor pipelines. Due to aggressive technology and voltage scaling, runtime errors are becoming increasingly common. The proposed techniques have been implemented for simple inorder pipelines as well as out-of-order superscalar pipelines. Results for inorder pipelines show an area overhead of around 15% and a 20% increase in clock cycle time. Our proposed techniques for front-end structures in superscalar pipelines, were able to detect, correct and recover from all single bit errors with an area overhead of 0.5%, power overhead of 2% and performance overhead of around 8%. These overheads are minimal compared to roughly 3 times overhead in area for Triple Modular Redundancy techniques.
Chapter 5

Soft Error Field Failure Analysis

Information systems are employed in an increasing number of application areas, motivating research activity focused on reducing the time to market, improving performance, reducing power consumption and increasing reliability. This latter aspect has assumed an increasingly important role, especially when dealing with critical applications.

Typically, present day information systems, contain multiple processor cores. As device geometries continue to shrink, the hardware components in the information systems, such as microprocessor, memory and ASIC devices will experience an increasing number of transient and permanent failures. As a consequence, future designs will need to be able to detect permanent hardware failures and recover from transient errors. Software failures (bugs) in the operating system and/or transaction processing software can also adversely affect overall system reliability [89]. The complexity of the systems has reached to a point where it is impossible to detect every error during the design phase. A number of errors escape to the field. These errors can potentially decrease reliability and availability of the systems.

When the effect of soft error is manifested at the system level, it is generally in the form of a sudden malfunctioning of the system, which cannot be readily attributed to a specific cause. Soft errors are un-traceable once new data has been written into the memory that stored corrupted bits or when the power is reset. Hence, it is hard to identify soft error
as the root cause of the problem. Furthermore, the problem is not reproducible due to its random nature. Because of this, it is usually very difficult to show that soft errors are causing the system failures.

In this part of the thesis, actual field data has been used to analyze runtime errors occurring in the 32-bit microprocessors used in high performance systems. This work is done based on collaboration with a major industrial information computing system manufacturer. It complements the work done in [7], which analyzed field data for computing SER in FPGA-based designs. As it is not possible to observe the internal structure of the microprocessor at a system level, determining soft error rates using field data for microprocessors is more challenging compared to performing the same study for FPGA-based designs.

The remainder of this chapter, is organized as follows:

A review of previous work done on analyzing failure data is given in Section 5.1. Some details about the information systems studied, are presented in Section 5.2. Section 5.3 details the methodology used in carrying out the study. Some examples showing how the data was bucketized are presented in Section 5.4. Section 5.5 attempts to obtain FIT rates due to SEUs as well as localization of SEUs within the microprocessor. The relative frequency and failure trends of Possible SEUs and other causes of runtime failures are discussed in Section 5.6. Section 5.7 presents a qualitative analysis of different trends observed in the study. A summary is presented in Section 5.8.

5.1 Related Work

In this section, we discuss prior studies that analyzed field failure data collected from error logs of information systems. In [90], 70 Windows/NT mail servers were observed over 6 months. Data was collected from error logs. However, since a Local Area Network was studied, most of the results related to failures of networking components.

Oppenheimer et al. studied error logs taken from 3000 machines on the Internet [91].
Six months of data was analyzed. Most of the results were related to failures linked to various Internet services.

In [92], data was collected from detailed observations of a large disk drive population in a production Internet services deployment. It was observed that very little correlation existed between failure rates and either elevated temperature or activity levels.

Jiang et. al. discuss trends occurring in a large set of real world customer cases reported from 100,000 commercially deployed storage systems [93]. The results showed that while some failures are either benign or resolved automatically, many others can take hours or days of manual diagnosis to fix. For modern storage systems, hardware failures and miscon-figurations dominated the customer cases. Also some software failures took a long time to resolve. The analysis further showed that a failure message alone is a poor indicator of root cause, and that combining failure messages with multiple log events can improve low-level root cause prediction by a factor of three.

A recent analysis of field failure data was done in [89]. The systems considered were comprised of compute nodes in a cluster. Six categories of failures were analyzed and probability distributions were identified for various classes of failure.

The susceptibility of commodity operating systems running on IA-32 and IA-64 micro-processors to soft errors was investigated in [91]. The results indicated that with improved microprocessor support like the MCA, and a little application knowledge, few of the detected soft errors needed to result in fatal system errors, especially because many of these bit flips would be overwritten in the future.

Radiation testing has been done on microprocessors in [95]. The impact of semiconductor technology scaling on neutron induced soft errors was studied using accelerated testing. Mean time to failure was obtained for a state-of-the-art microprocessor running Matrix benchmarks.
5.2 Information Systems

The information systems under consideration in this work are high-availability systems. These systems typically hold several hundred disks which can be protected via RAID protocols. The internal bussing architecture provides for a high degree of redundancy so that a failure in any component on a link does not disconnect other components from the system. However, simultaneous failures in multiple components can lead to events which impact reliability and potentially reduce availability. We refer to these as Reliability Impact Events (RIEs). Such events may cause sudden interruptions in service and potentially result in data unavailability. Our analysis is based on studying such events in the error logs of live systems.

One of the components that interconnect to the buses is the Logical Unit Module (LUM). Each LUM contains one or two single-core state-of-the-art IA-32 microprocessors (the so-called server processors), depending on the particular system. They are responsible for processing I/O requests arriving through the host. Two levels of on-chip caching are available on these microprocessors. The L1 data cache is parity protected (single-bit errors can be detected but not corrected), while the L2 cache is protected using Error Correcting Codes (ECC) for data and parity for tags. ECC circuitry is able to detect and correct single bit errors.

In this study, the following two classes of systems are considered:

System Type A:- In these systems, there are two server microprocessors on each LUM and two LUMs per system, for a total of four microprocessors per system. Two levels of on-chip caching are available on these microprocessors. The L1 data cache is parity protected, while the L2 cache has ECC protection on data and parity protection on tags. The processors used in System A, are superscalar and have a 20-stage pipeline.

System B:- Similar to Systems A, these information systems have two LUMs in each system. We break down the machines included in Systems B into two subclasses (Systems
B1 and Systems B2). While there are two processors per LUM in System B1, System B2 contains only a single processor per LUM. All other aspects of Systems B1 and B2 are exactly the same.

5.2.1 Machine Check Architecture

The microprocessors in the systems under study are equipped with a Machine Check Architecture (MCA) as described in [96] and [97]. The MCA allows the operating system to detect, signal, and record information about selected machine fault conditions, like parity errors, division-by-zero etc. Some of the faults are correctable, while others are uncorrectable (i.e., only detectable).

5.3 Methodology

In this Section, a description of the methodology employed to analyze the field data is presented.

5.3.1 SEUs in the field

In the information systems studied in this work, we limit our focus on Reliability Impact Events (RIEs) occurring in the microprocessor. Such events did not result in LUMs being replaced in the system.

Failure logs were obtained from the repository of field data available from the manufacturer. The methodology used to identify possible SEUs is shown in Figure 5.1 and is outlined below:

Initially, all unscheduled RIEs occurring in the micro-processors contained in the LUMs were analyzed. A large number of unscheduled RIEs were identified. These can occur for one of several possible reasons:

• Component failure


Select an unscheduled RIE from the database

RIE due to Component Failure?

Yes → Classify as Non-SEU event

No

RIE due to Software Failure?

Yes

No

RIE due to Power Issues?

Yes

No

Classify as Possible SEU event

Figure 5.1: A flowchart of the methodology

- Software-based errors
- Power issues
- Radiation-induced errors

The next step was to isolate those cases that could have occurred because of a soft error. We call these *Possible* soft errors. Errors that were confirmed to have occurred inside the microprocessor using error logs and traces were labeled as *Probable* soft errors and are denoted by $SEU_{CPU}$. Most of these cases resulted in Machine Check exceptions. In some cases, we were able to identify the register in which a bit flip had occurred, with the help of error traces.

In some cases invalid memory addresses were generated by the microprocessor and resulted in a microprocessor RIE. If an RIE could not be linked to faulty software, and the traces did not point to a specific register in which the error occurred, we refer to these cases
as $SEU_{INV-MEM}$. Since invalid addresses may still be generated because of a software bug, we do not include them in the Probable soft error bucket. Instead, we construct a Potential soft error bucket and include such cases in this category.

The Potential soft error bucket also contains those cases of isolated RIEs in which the logs did not point to a specific fault in hardware or software. The error could not be confirmed to have occurred in the microprocessor or be attributed to an invalid memory address. Most of these errors were classified as Potential because some of the logs that were studied were incomplete. However, since these were isolated cases out of the total number of RIEs, they were potential candidates for SEUs. We include these cases in a category called $SEU_{no-info}$. Our classification taxonomy is shown in Figure 5.2.

The following approach was taken to identify these cases:

1. Note the time and date of the RIE.
2. Search the events in the microprocessor logs, for that incident.
3. If the events preceding the incident can be classified as one of the reasons mentioned above, do not include them in the Possible SEU bucket.

Using the above criteria, we obtained a bucket containing Possible soft errors that had a significant number of incidents. Note that these are individual instances of RIEs that could not be linked to a hardware component, power supply issue, or corrupted software.
5.3.2 Localizing SEUs within the microprocessor

Probable SEUs occurring inside the microprocessor were further investigated with the help of the Machine Check Architecture (MCA) information available. This information helped in isolating just exactly where in the microprocessor the SEU occurred. The exception handler returns a 64-bit error code. The error code helps to identify where in the cache hierarchy the error occurred. Bits 0 to 15 contain the machine check error code, and bits 16 to 31 contain the model-specific error code. There are some reserved and informational bits and also some flags. The error code helps in finding out where (L1 cache, registers etc) the error occurred inside the microprocessor.

The format of the 16-bit error code is as follows: 0000 0001 RRRR TTLL. The description of the fields is as follows.

- The 2-bit TT sub-field indicates the type of transaction: data (00), instruction (01), or generic (10). A generic type is reported if the microprocessor is unable to determine the type of the transaction.

- The 2-bit LL sub-field indicates the level in the memory hierarchy where the error occurred: level-0 (00), level-1 (01), level-2(10), or generic (11). Again, the generic type is reported when the micro-processor cannot determine the hierarchy level.

- The 4-bit RRRR sub-field indicates the type of the action associated with the error. Actions include read and write operations, prefetches, cache evic-tions, and snoops.

5.3.3 Analyzing failure trends

All of the incidents related to a particular root cause were selected from the logs. A sample was constructed from each class of failure and the install dates of the systems in the sample were determined. This gave us the time-to-failure for each system in units of days. Based on this data, we identified candidate failure distributions and constructed probability plots for each class of failure. Since multiple distributions may fit the data, we try to fit more
than one distribution to the failure data. To determine the best fit, a comparison of $R^2$ values (i.e., goodness of fit) is an objective criterion. This value may be obtained by fitting a straight line through the data points and using the linear equation of that line. An $R^2$ value of 1, means that the distribution fits the data exactly. The farther the value of $R^2$ is from 1, the poorer the distribution fits with the data. The following distributions were considered:

- Normal Distribution: This is the most widely used distribution. It generally requires a significantly large number (typically greater than 30) of sample points to generate an accurate estimate of failure rates. It is used to represent variability in manufactured goods, wear out effects and a host of other phenomena. If the normal distribution provides a good fit to the data, the Reliability at time $t$, can be estimated using the formula:

$$R(t) = 1 - \Phi \left( \frac{t-\mu}{\sigma} \right)$$

where, where $\mu$ and $\sigma$ are mean and standard deviation of the sample, respectively.

- Lognormal Distribution: This distribution is widely used in reliability engineering to describe failure caused by fatigue, uncertainties in failure rates and other phenomena. In [98], it is suggested that lognormal distributions are the most appropriate distributions to fit failure data for software. Reliability can be computed using the formula:

$$R(t) = 1 - \Phi \left( \frac{\ln(t/t_0)}{w} \right)$$

where $t_0$ and $w$ are parameters of the lognormal distribution which can be estimated from the line fitting the data. If $a$ is the slope of the line and $b$ is the y-intercept, then $w$ and $t_0$ can be estimated by:

$$w = \frac{1}{a}, \quad \text{and} \quad t_0 = e^{-\frac{b}{a}}$$

- Weibull Distribution: Weibull distribution is widely used for describing times to failure and the strength of brittle materials. In situations where the underlying distribution
is not explicitly known, but the failure arises from many competing flaws, the Weibull distribution often provides a good empirical fit to the data. It provides a more accurate estimate than a Normal distribution if the number of data points is small. The Reliability can be estimated using the formula:

\[ R(t) = e^{-\frac{t}{\theta}} \]

where, \( \theta \) and \( m \) are parameters of the Weibull distribution that can be estimated from the line fitting the data. If the slope of the line is \( a \) and the y-intercept is \( b \), the parameters \( \theta \) and \( m \), can be estimated by:

\[ m = a, \quad \theta = e^{\frac{b}{a}} \]

- Exponential Distribution: This distribution is employed when constant failure rates adequately describe the behavior of continuously operating systems. For a device described by a constant failure rate, the probability that it will fail during some period of time in the future is independent of its age. Exponential distributions have been used to obtain parameters such as Mean time between Drops in telecommunication systems \[99\]. Reliability is given by:

\[ R(t) = e^{-\frac{t}{\sigma}} \]

\( \theta \) can be estimated using the slope of the line fitting the data using the relation \( \theta = \frac{1}{a} \).

### 5.4 Examples

In this section, we will present four examples to show how an isolated RIE can point to a soft error.

#### 5.4.1 A Bit flip in the Instruction Cache

For a particular RIE, a machine check exception was generated and the error code returned was 0xA200000084010452. If we inspect the last 16 bits (0000 0100 0101 0010), we see that the 2 rightmost bits (10) denote that the error occurred in the L2 cache. The next 2 bits
(00) indicate that the error was in the instruction cache. Code 0101 translates to a read error. Hence, we can conclude that a parity error occurred in the L2 instruction cache.

5.4.2 A Bit flip in a function address

In another incident, an attempt was made to access an invalid memory address that resulted in a RIE. Stack traces were uploaded and the logs pointed to the following events prior to the incident:

- The last call before the incident, references the address 0xf9a96652. This is an actual function pointer, so this seems to be a valid value. Looking at the lines that should have been executed we find:

  FunctionABC: 0xf9a96652 55 push ebp

  The value at this address translates to push ebp. However, the value on the stack is not ebp.

- The address where the system incurred this error was 0xf9a96258, which is not in the function.

- If we look at the instructions preceding the instruction where the system halted, we find:

  0xf9a96252 53 push ebx
  0xf9a96253 183b sbb [ebx],bh
  0xf9a96255 55 push ebp
  0xf9a96256 d476 aam
  0xf9a96258 0884c90f851304 or[ecx+ecx*8+0x413850f],al

  The values in ebx and ebp match exactly the values on the stack. The last instruction at 0xf9a96258 referenced an invalid pointer that resulted in an RIE.
• We can see here that instead of going to \texttt{0xf9a96652}, the microprocessor went to the address \texttt{0xf9a96252}, which is a difference of 1 bit (the 11th bit from the right). Since the error is confirmed to have occurred inside the processor, we include this incident in the set of Probable SEUs.

### 5.4.3 A Probable SEU in the Instruction Pointer

We now examine an RIE that occurred in the microprocessor because of an invalid memory address. With the help of error traces, we were able to identify the events leading up to the incident. The following is the sequence of events prior to the incident.

- Function \textit{ABC} was called just before the RIE occurred.
- Before calling \textit{ABC}, the return address was pushed on the stack. The calling function was in the range \texttt{0xa5be8xxx - 0xa5beexxx}
- The value of the Instruction Pointer (IP) at the time of RIE was \texttt{0xa7be8a8e}. This was an invalid address that resulted in the incident being signaled.
- Changing the IP from \texttt{0xa7be8a8e} to \texttt{0xa5be8a8e}, which lies within the range of the calling function, we can see that an invalid value was popped off the stack on the return from the function \textit{ABC}. The true value differed from the actual value by one bit (the 7th bit from the left), and this was not identified to be a software bug. The incidence was included in the probable SEU bucket because the error was confirmed to have occurred inside the processor.

### 5.4.4 A Potential SEU due to invalid memory address

In one instance of an RIE, an invalid memory address was generated which led to a memory access violation. A software bug is generally the most probable cause of invalid memory accesses. However, in this RIE incident, there was no subsequent memory violation, even
though the software running on the machine was not changed or upgraded. Since, the error was not reproducible and did not occur because of a specific software or hardware issue, we included it in the Potential SEU bucket in the $SEU_{INV-MEM}$ category.

5.5 Analysis of SEU Events

By looking at error logs and error traces, and with the help of Machine Check Architecture, we tried to bucketize the incidents into Possible, Probable and Potential SEUs. We also compute the FIT rates due to SEUs in microprocessors under study.

5.5.1 SEU distribution in the systems

There were a significant number of unscheduled RIEs that occurred in these systems. Over a hundred of these were classified as Possible SEUs. Figure 5.3 shows the statistics of Possible SEU events occurring in the two systems.

![Figure 5.3: Distribution of SEUs in (a) System A (b) System B](image)

It was observed that some of the RIEs were due to events occurring inside the CPU ($SEU_{CPU}$). Other events were due to invalid memory addresses generated by the microprocessor ($SEU_{INV-MEM}$). Since these invalid addresses resulted in a single instance of an RIE and were not linked to faulty software, they were included in the Potential soft error bucket. A similar number of indeterminate cases were found where there was an isolated
instance of an RIE, though it could not be linked to faulty hardware or software. In some cases, there was insufficient information in the logs, but since these were single instances of an RIE, and they were not linked to any software or hardware failure, they were included in the Potential SEU bucket. We group them as $SEU_{NO-INFO}$ in Figure 5.3.

5.5.2 SEU localization within the microprocessor

When an SEU is confirmed to have occurred inside the microprocessor, we classify it as a Probable SEU. For systems of type A, we analyzed a statistically significant number of probable SEUs and identified the exact location of SEUs occurring within the microprocessor using the Machine Check Architecture (MCA) information [100, 96]. The MCA identifies errors occurring in the cache hierarchy, as well as those occurring in other structures like TLBs, External Bus unit etc. Additionally, some invalid memory accesses, which were traced back to bit flips in the processor registers, were also included in the probable SEU list.

The L1 cache in the micro-processor under study has parity protection. The L2 cache is an 8-way set associative cache with ECC protection on the data array and parity protection on tag array. As mentioned in [32], ECC protection reduces the effective vulnerability of the L2 data array to zero and the primary source of L2 vulnerability is the tag array. It was observed that the L2 tag vulnerability is greater than the vulnerability of L1 data and Instruction cache for most of the SPEC benchmarks when a similar processor and memory organization was considered. To further investigate the vulnerability of the L2 tag ad-dresses, the L2 tag vulnerability was profiled in terms of pseudo-hit vulnerability, replacement vulnerability, multi-hit vulnerability, and status vulnerability in [32]. The results show that replacement vulnerability makes up almost 85% of the total tag vulnerability. This is due to the fact that tag addresses become more susceptible once the first write occurs in the block and remain susceptible to SEUs until the block is replaced or flushed to lower levels of memory. The field analysis carried out in this work agrees with the findings presented in [32]. With the help of the MCA, it was observed that only 12% of the errors were found to have occurred
in the L1 cache. This is primarily due to its small size. Almost 80% of the probable SEUs occurred on tag bits of the L2 cache. Additionally, there were some parity errors in the interconnect buses (which are only parity protected). There were no errors found in the other structures using the MCA. The results are presented in Table 5.1.

<table>
<thead>
<tr>
<th>Parity on L2 Tag</th>
<th>80%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Parity on L1 Cache</td>
<td>12%</td>
</tr>
<tr>
<td>Interconnects</td>
<td>8%</td>
</tr>
</tbody>
</table>

5.5.3 Calculating FIT rates due to SEUs

The FIT rates were calculated separately for probable SEUs and all possible SEUs. They were obtained by dividing the number of SEUs by total run time of all systems. Running time was computed using the following formula:

\[
\text{running time} = \sum_i \text{system}_i \times \text{time in field}_i
\]

for \( \text{system}_i \) in the field for \( \text{time in field}_i \).

In this study, more than half a billion system-hours of in-formation systems in the field were analyzed. System A has significantly higher systems-hours than System B. Table 5.2 shows the relative FIT rates (to the Probable SEU FIT rate of system A) for these systems based on Probable and Possible SEUs (i.e., the lower and upper bounds). Due to extreme sensitivity of such data, we have only reported the relative values instead of the actual FIT rates in this paper.

<table>
<thead>
<tr>
<th></th>
<th>Relative FIT (Probable SEU)</th>
<th>Relative FIT (Possible SEU)</th>
</tr>
</thead>
<tbody>
<tr>
<td>System A</td>
<td>1</td>
<td>3.68</td>
</tr>
<tr>
<td>System B</td>
<td>3.24</td>
<td>8.03</td>
</tr>
</tbody>
</table>

Table 5.1: Statistics for Machine Checks in Systems A

Table 5.2: Relative FIT rates for Systems A and B.
5.6 Comparison of SEU and non-SEU events

In this section, we compare the relative frequency and failure trends of Possible SEUs and other causes of runtime failures.

5.6.1 Failure Distributions

We further break down the machines included in Systems B into two subclasses (Systems B1 and Systems B2). While there are two processors per LUM in System B1, System B2 contains only a single processor per LUM. The processors used are exactly the same in Systems B1 and B2. We studied the relative frequency of Possible SEUs compared to other causes of RIEs for these systems. The results for each system type as well as total RIE cases are presented in Table 5.3. The results for each column (representing each system type) are normalized to the number of Possible SEUs for that system. The graphical representation of this data is shown in Figure 5.4.

<table>
<thead>
<tr>
<th>Source of RIE</th>
<th>System A</th>
<th>System B1</th>
<th>System B2</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hardware related</td>
<td>1.91</td>
<td>2.27</td>
<td>7.25</td>
<td>2.19</td>
</tr>
<tr>
<td>Power related</td>
<td>0.18</td>
<td>0.19</td>
<td>0.5</td>
<td>0.19</td>
</tr>
<tr>
<td>Software related</td>
<td>2.41</td>
<td>4.44</td>
<td>18.12</td>
<td>3.48</td>
</tr>
<tr>
<td>SEU related</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

RIEs can occur due to several reasons. The first three entries in Table 5.3 are Non-SEU related, whereas the last entry is SEU-related. These categories can be explained as follows.

- **Hardware-related failures**: These failures are permanent faults attributable to a faulty hardware component. For instance, the hardware of the microprocessor may itself be defective. This is observed when there are multiple machine checks in a short period of time. Sometimes faults on other components connected with the microprocessor (associated components) may cause RIEs.
• Power-related failures: Although the information system has spare power supplies, power spikes and failures can result in RIEs. If the events preceding the RIE point to an unstable power supply, the incident is classified as a Power-related failure.

• Software-related failures: A large percentage of the incidents occurred because of software errors. This includes the software running on the host (e.g., the load balancing software) and the operating system running on the microprocessor. RIEs can also occur when the system is improperly configured at installation time or during an upgrade.

• SEU-related failures: These temporary failures correspond to all Possible SEUs. This category contains all RIEs due to Probable and Potential SEUs.

Hardware-related failures are permanent, whereas the other three types of failures are
temporary. It can be observed that power-related failures constitute a very small percentage of the total RIEs. It needs to be mentioned that although software related failures correspond to the largest percentage of RIEs, they can be fixed using error traces and error logs.

5.6.2 Severity Analysis

Every RIE is characterized by a high, medium or low severity, by field engineers, depending upon its availability impact and the availability requirements of the particular system in the field. The results are graphically depicted in Figure 5.5. Each bar in the figure represents the severity levels of various causes of RIE, as described above. The number in the brackets is the average severity level. The severity values ranges from 1 (low) to 3 (high). It can be observed from Figure 5.5, that severity of various sources of failures is system dependent. For newer systems (systems of type B1 and B2), software related failures are the most severe. For more stable systems (type B2) with more developed software, hardware-related failures are most severe. On average (see Figure 5.5(d), hardware-related failures are more severe than transient failures. The average level of severity of SEU-related RIEs is slightly smaller as compared to other transient causes. This is because SEUs can result in isolated instances of RIEs and that the machine works normally after RIEs occur (i.e., no system errors are a direct result of SEUs occurring). In summary, SEUs are associated with the least severe RIEs in current systems. However, as the frequency of soft errors exponentially increases with each new technology generation, the severity of SEU-related failures is expected to increase accordingly.

5.6.3 Failure trend analysis

Probability plotting is an extremely useful technique to estimate the distribution parameters. It provides both a graphical picture and a quantitative estimate of how well the distribution fits the data [99]. We collected a sample of 100 distinct systems and obtained
CHAPTER 5. SOFT ERROR FIELD FAILURE ANALYSIS

Figure 5.5: Severity of RIEs due to various causes in different systems
the time-to-failure based on the install date and the failure event. The sample was constructed based on the frequency of various types of failures. Table 5.4 shows the number of incidents in the sample, based on the root cause of the incident.

Table 5.4: Distribution of systems in the sample

<table>
<thead>
<tr>
<th>Cause of Failure</th>
<th>Number of Systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hardware</td>
<td>42</td>
</tr>
<tr>
<td>Software</td>
<td>35</td>
</tr>
<tr>
<td>Transient</td>
<td>18</td>
</tr>
<tr>
<td>Power</td>
<td>5</td>
</tr>
</tbody>
</table>

**Hardware Failures**

The CDF of distributions were obtained for hardware failures as mentioned in Section-sec:casemethod. The $R^2$ values for different probability distributions are shown in Table 5.5. The best $R^2$ value (i.e., closest to 1.0) among all the probability distributions is 0.9629 for the Normal distribution.

Table 5.5: $R^2$ values for Hardware Failures

<table>
<thead>
<tr>
<th>Distribution</th>
<th>$R^2$ values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal</td>
<td>0.9629</td>
</tr>
<tr>
<td>Lognormal</td>
<td>0.8436</td>
</tr>
<tr>
<td>Weibull</td>
<td>0.9508</td>
</tr>
<tr>
<td>Exponential</td>
<td>0.7398</td>
</tr>
</tbody>
</table>

Figure 5.6(a) shows the plot with the data fitted to a Normal distribution.

The plot of reliability against time is shown in Figure 5.6(b). It can be observed that Reliability is decreasing with time. If the distribution fitted the data more closely, Reliability would be exactly equal to 1 at time 0.
Software Failures

The CDF of the distributions were obtained as before. The $R^2$ values obtained for various distributions are given in Table 5.6. The best $R^2$ value obtained is 0.9604 for the Normal distribution, which is plotted in Figure 5.7(a). Some researchers have suggested that software faults follow a lognormal behavior. However, as observed in Table 5.6, the data does not fit well to a lognormal distribution.

Figure 5.7(a) shows the plot with the data fitted to a Normal distribution.

The Weibull and Exponential distributions produced a worse fit than the Normal and Lognormal distributions. The plot of Reliability versus time for Software is shown in Figure
Table 5.6: \( R^2 \) values for Software Failures

<table>
<thead>
<tr>
<th>Distribution</th>
<th>( R^2 ) values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal</td>
<td>0.9604</td>
</tr>
<tr>
<td>Lognormal</td>
<td>0.7749</td>
</tr>
<tr>
<td>Weibull</td>
<td>0.9086</td>
</tr>
<tr>
<td>Exponential</td>
<td>0.7142</td>
</tr>
</tbody>
</table>

Power-related failures

The CDF of the distributions are obtained as before. Since the number of data points was small, we did not plot the data for the Normal and Lognormal distributions as they typically need 25 to 30 data points. The \( R^2 \) values obtained are shown in Table 5.7(b).

Table 5.7: \( R^2 \) values for Power Failures

<table>
<thead>
<tr>
<th>Distribution</th>
<th>( R^2 ) values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weibull</td>
<td>0.9882</td>
</tr>
<tr>
<td>Exponential</td>
<td>0.8834</td>
</tr>
</tbody>
</table>

The best \( R^2 \) value obtained is 0.9882, which shows that the data fits well with a Weibull distribution. The probability plot is shown in Figure 5.8(a). The values of \( a \) and \( b \) obtained from the equation of the fitted line are 5.09 and -29.216, respectively. This gives a shape parameter \( m \) equal to 5.09 and scale parameter \( \theta \) equal to \( \exp^{29.21/5.09} = 311 \). The Reliability is plotted against time in Figure 5.8(b).

Transient faults

The CDF of the distributions is obtained as before. Since these failures are transient and unpredictable, they do not follow any particular distribution very well. We present the results for the Weibull and Normal distributions in Figure 5.9(a) and 5.9(b), respectively. The \( R^2 \) values are given in Table 5.8.
Figure 5.7: Probability and Reliability plots for Software failures

The plot of Reliability against time is shown in Figure 5.10. Note that in case of Transient failures, the Weibull distribution did not fit the data very well compared to some other classes of failures. Hence, the shape of the reliability curve is slightly different compared to other classes of failures.

5.7 Analysis of Results

The FIT rate analysis of the microprocessor used in these systems, as presented in Table 5.2 suggests that the FIT rates of the microprocessors in System B are significantly (2-4
CHAPTER 5. SOFT ERROR FIELD FAILURE ANALYSIS

Figure 5.8: Probability and Reliability plots for Power failures

Figure 5.9: Weibull (a) and Normal (b) Probability plots for Transient Failures
Table 5.8: $R^2$ values for Transient Failures

<table>
<thead>
<tr>
<th>Distribution</th>
<th>$R^2$ values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal</td>
<td>0.9071</td>
</tr>
<tr>
<td>Lognormal</td>
<td>0.8954</td>
</tr>
<tr>
<td>Weibull</td>
<td>0.9234</td>
</tr>
<tr>
<td>Exponential</td>
<td>0.7746</td>
</tr>
</tbody>
</table>

Figure 5.10: Reliability against time for Transient Failures

times) higher than System A. A look at the architectures of Systems A and B, as presented in Section 5.2 shows that the on-chip cache of System B is twice the size of the cache in System A. Although the computed FIT rates are limited to particular RIE events and analysis of error logs, a comparative analysis can factor out common sources of error in the analysis. This data suggests that the system-level FIT rate of server microprocessors in this study has a direct relationship to the size of on-chip cache memory. Moreover, the distribution of SEU and non-SEU related failures, as presented in Figure 5.4 shows that the percentage of SEU-related failures in systems of type A and B1 are almost the same and almost twice of that in type B2. As mentioned in Section 5.2, the systems of type A and B1 have the same number of microprocessors per system (4) which is twice of that in type B2 (2). Therefore, this data suggests that the percentage of SEU-related failures in the system is proportional to the number of microprocessors in the system, as expected. It can be observed from the results in the preceding section, that permanent
(hardware and software) failures follow the normal distribution pretty well. Since we have fewer data points for Power and transient failures, the Weibull distribution is giving a good approximation in these cases. As mentioned in [98], in many cases the software application failures follow log-normal distribution. However, in our case the normal distribution gave a better fit. One of the reasons might be that in the systems under study, software is mainly used as a transaction processing tool rather than a computation intensive tool. So the failures resulting from software malfunction mainly result in lower rates of transaction processing rather than causing catastrophic failures. Since transient failures are mostly due to environmental factors (rather than wear-out or bugs), they are not following any patterns very closely. The Reliability plots shown in Section 5.6 give an insight into the failure probabilities. The plots can be used for example, to determine the warranty periods of the systems. Since separate Reliability plots are obtained for different classes of failures, warranty periods of systems against various classes can be predicted as well.

It is important to note here, that in many instances complete logs were not uploaded at the time when RIE occurred. This prevented us from finding the root cause of those events. In some cases, the customer was not interested in an isolated RIE instance for example, and did not ask for finding out the root cause. Additionally, since the process of uploading the logs in the database is semi-manual, there are some human errors in the process. If the process is fully automated, the logs would be much more complete and hence easier to analyze.

5.8 Summary

We performed a case study based on field failure data involving commonly used 32-bit server processors in high performance systems. Our attempt was to classify the relative occurrence of soft errors compared to other runtime errors. It was observed that the SEUs occurring within the microprocessors are mostly affecting the tag array of the caches. As
tags are only parity protected in most microprocessors, the SER is dependent on the size of the cache. Additionally, we also considered the system effects of non-SEU events such as permanent faults and software failures. A comparison of the reliability impact of SEU versus non-SEU events was also performed, based on the potential impact to customers. The field failure data for various causes of errors was also plotted for different known probability distributions. We found that the error trends found in particular memory structures inside the microprocessor agree with results obtained in earlier work using analytical models.
Chapter 6

Conclusions

In electronic circuits, data bits are represented by small packets of charge. When these charge packets are modified by spikes, noise or radiations, the stored information may get changed. As process technologies continue to shrink, there has been an increase in the number of radiation induced errors in live systems. This has led to serious reliability challenges in the design phase.

In this thesis, we propose high level modeling and mitigation techniques for handling transient errors in present day systems. Using SAT-based methodologies, we attempt to obtain accurate Soft Error Rates for combinational and sequential circuits described at the behavioral level. Extensive simulations on medium and large sized benchmark circuits show the scalability of the proposed techniques in terms of circuit size and runtime.

We also used SAT-based concept to compute a metric called the \textit{Hardware Vulnerability Factor}, that captures the microarchitecture vulnerability of processor structures, irrespective of the code running on the processor. The technique was tested on two inorder processor pipelines and HVF of various stages of the pipeline were obtained and compared.

Several error detection, correction and recovery techniques for handling soft errors in simple and superscalar pipelines were presented. The techniques introduce minimal area and performance overhead. Implementations were done at both the RT-level and performance
simulator level. Extensive fault injections were performed at both the levels. Results for SPEC2000 benchmark circuits are presented.

Finally, we discuss and interpret the results of a case study, conducted on thousands of live high-availability systems in the field. The field analysis looks at the relative occurrence and severity of soft error related incidents compared to other permanent and temporary errors. Reliability estimates were obtained by analyzing the trends of these failures using probability plots.

As future work, several directions can be taken.

- Extending the SER estimation methodology: The SAT-based methodology for obtaining soft error rates at behavioral level, can be extended to larger combinational and sequential circuits with the advent of faster and more powerful SAT solvers. Additionally, SER computation in circuits with feedback loops need to be investigated further.

- Error detection and recovery for other structures: The methodologies for error detection, correction and recovery in superscalar pipelines can be modified to cover other portions of the processor pipeline as well as other architectures like the x86 and SPARC. Methodologies for handling errors occurring at the back-end would also be useful.

- Uncore logic: As the number of cores in a chip increases, the uncore logic is becoming more and more complicated. Efficient error detection and recovery methods need to be devised for often unprotected, uncore logic as well.

- Post-silicon debug: Post-silicon debug is becoming an integral part of the design cycle. This step involves detecting the error, isolating the error and fixing the error as it occurs during normal circuit operation. The methodologies present in this thesis for error detection, correction and recovery can be applied at the post-silicon debug
stage, as the exact location of errors can be determined and correction schemes can be used for fixing the errors.
Bibliography


[17] “Cisco 12000 single event upset failures: Overview and workaround,”

[18] C. Weaver, J. Emer, S. S. Mukherjee, and S. K. Reinhardt, “Techniques to reduce the
soft error rate of a high-performance microprocessor,” in Proceedings of the Interna-


E. Schwarz, and M. Vaden, “Ibm power6 microarchitecture,” IBM Journal of Re-

and recovery,” IEEE Transactions on Device and Materials Reliability, vol. 5, no. 3,

[23] R. Baumann, “Soft errors in advanced semiconductor devices-part i: the three radi-
ation sources,” IEEE Transactions on Device and Materials Reliability, vol. 1, no. 1,

[24] G. Srinivasan, “Modeling the cosmic-ray induced soft-error rate in integrated circuits:
an overview,” IBM Journal of Research and Development, vol. 40, no. 1, pp. 77–89,
1996.


[37] “Relsat 2.1, [http://www.bayardo.org/resources.html](http://www.bayardo.org/resources.html)”

[38] “Cadence smv symbolic model checker, [http://www.kenmcmill.com](http://www.kenmcmill.com)”


[88] “The m5 simulator,” [http://www.m5sim.org](http://www.m5sim.org).


