Online Low-Cost Defect Tolerance Solutions for Microprocessor Designs

by

Kypros Constantinides

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2009

Doctoral Committee:

Associate Professor Todd M. Austin, Chair
Associate Professor Achilleas Anastasopoulos
Associate Professor Scott Mahlke
Assistant Professor Valeria M. Bertacco

© Kypros Constantinides 2009
All Rights Reserved

To my family

ACKNOWLEDGEMENTS

First, I would like to thank my advisor Prof. Todd Austin for his support and guidance that made the completion of this dissertation possible. Over the last five years, under the supervision of Prof. Austin, I had the opportunity to learn a lot from him and acquire the necessary abilities to carry out independent research. During my graduate studies, in multiple occasions where I felt I was loosing focus and direction, Prof. Austin was a constant source of inspiration and guidance that always put me back on truck towards the pursue and completion of my research goals. I feel that my interaction with Prof. Austin provided me with several skills, values, and knowledge that would be very helpful in my future career and life, and I am deeply grateful to him for that.

I would also like to thank the members of my dissertation committee: Prof. Valeria Bertacco, Prof. Scott Mahlke, and Prof. Achilleas Anastasopoulos. First, I appreciate their kindness to serve as members of my dissertation committee and I would like to thank them for the positive feedback, guidance, and support that they provided to me along the process. In particular, I would like to thank Prof. Bertacco that together with my advisor Prof. Austin supervised and supported me for most of my dissertation work. Her help and support was an important cornerstone for the completion of this dissertation. Early in my graduate studies I also enjoyed working with Prof. Mahlke in multiple projects that helped define my dissertation topic.

I am also deeply grateful to all the people that I had the opportunity to collaborate with on many of the projects presented in this thesis: Smitha Shyam, Sujay Phadke, Mojtaba Mehrara, Mona Attariyan, Jason Blome, Stephen Plaza, Bin Zhang, Michael Orshansky, Andrea Pellegrini, Dan Zhang, and Shobana Sudhakar. I would also like to thank my lab-mates, both current ones and those that already graduated and left the lab, for the great working and research environment that they provided: Andreas Moustakas, Joe Greathouse, Jason Clemons, David Ramos, Ilya Wagner, David Meisner, Andrew DeOrio, Leyla Nazhandali, and Michael Minuth.

I am thankful to my mentor from Microsoft Research, Onur Mutlu, for his support and guidance during my graduate studies. My collaboration with Onur started when I interned at Microsoft Research at the summer of 2007 and since then he is been a constant source

for advice that I could rely on. Over the last two years, my discussions with Onur shaped in many ways the work that is presented in this thesis.

I would also like to thank Prof. Yiannakis Sazeides for the early influence that he had on me during my undergraduate studies. While working with him during my undergraduate studies I had the chance to participate for the first time in research-oriented projects. That experience, fueled my interest in computer architecture, and later on, led to my decision to purse a Ph.D. degree in the area of computer architecture. I am deeply grateful to him for all the time that he spent with me in the lab teaching me how to carry out independent research, and for making me believe in myself that I could continue my graduate studies in the States.

Finally, I would like to thank my entire family for their support during my graduate studies. In particular, I would like to thank my parents for always encouraging me to pursue higher education and for raising me up with the values that I needed to complete this dissertation.

PREFACE

One of the major driving forces of the semiconductor industry is the continuous scaling of the silicon process technology. Over the last four decades, the scaling into a new silicon technology every few years offered to the computer architects smaller, faster, and cheaper transistors that made possible the development of high-performance microprocessors. This technological achievement also fueled the widespread adoption of microprocessor-based products in applications that touch every aspect of our life. However, the challenges in producing reliable devices in extremely dense silicon technologies are growing, with many device experts warning that continued scaling will inevitably result in silicon technology generations that are much less reliable than the current ones. Microprocessors manufactured in future silicon technologies will likely experience failures in the field due to silicon defects occurring during system operation. In the absence of any viable alternative technology, the success of the semiconductor industry in the future will depend on the creation of cost-effective mechanisms to tolerate silicon defects in the field while the microprocessor is in operation.

This thesis is focused on the exploration and evaluation of new alternative defect tolerance techniques that will provide low-cost online mechanisms to protect a microprocessor design from silicon defects. The approach of these novel defect tolerance solutions represents a new thinking in the field of defect-tolerant design. In particular, traditional approaches to defect-tolerant design saddle a system with costly redundant components that continuously verify the integrity of all computation. In contrast, the BulletProof approach, presented in this thesis, provides very low cost defect-tolerance through periodic online hardware checking by combining area-frugal hardware checkers with microarchitectural checkpointing. The use of checkpointing and recovery mechanisms provides computational epochs and a substrate for speculative unchecked execution. At the end of each epoch, the epoch's speculative computation is validated by checking the integrity of the underlying hardware using on-chip hardware checkers. This enables a low overhead solution that only needs to periodically check the integrity of the underlying hardware rather than continuously validate the execution using redundant computation.

To further lower the cost of the BulletProof mechanism and provide more flexible hardware checking strategies a new defect-tolerance approach is developed, called the Access-Control Extension (ACE) Framework, that shifts the silicon defect detection and diagnosis process from hardware to software. This new approach, allows special ISA instructions to access and control virtually any part of the processor's internal state. Based on this framework, special firmware periodically suspends the processor's execution and performs high-quality testing of the underlying hardware to detect defects.

This thesis, also makes the case that the hardware used to implement defect tolerance solutions, like the hardware resources of the ACE framework, can also be used for other applications to amortize their cost and ease the adoption of defect-tolerance mechanisms in future generation microprocessor designs. Specifically, it is demonstrated that the ACE framework hardware resources can also be used for (i) the online detection of design bugs, (ii) as a post-silicon debugging tool, and (iii) for improving the manufacturing testing process.

Finally, this thesis presents CrashTest, a novel FPGA-based framework used to assess the threats and the reliability requirements of a microprocessor design. The CrashTest framework differs from other resiliency analysis tools in two ways. First, it can automatically orchestrate a fault injection and analysis campaign on the gate-level netlist of a microprocessor design using an extensive collection of low-level fault models, and second, it employs FPGA-based accelerated hardware emulation to enable a detailed low-level failure analysis of complex full-system designs that can boot an operating system and run applications.

Altogether, the defect tolerance solutions presented in this thesis provide to a microprocessor design the same reliability guarantees as traditional defect tolerance techniques, but at a much lower cost and with higher flexibility and online adaptivity. This cost-effective defect-tolerance framework makes possible the development of reliable microprocessors using unreliable silicon technologies. The ability to use unreliable silicon technologies to manufacture reliable microprocessors will enable the continued silicon process scaling into smaller but less reliable transistors, a key requirement for the development of the next generation microprocessors and the extension of microprocessor-based products into new applications.

TABLE OF CONTENTS

DEDICATION . . . . **ii**

ACKNOWLEDGEMENTS . . . . **iii**

PREFACE . . . . **v**

LIST OF FIGURES . . . . **x**

LIST OF TABLES . . . . **xiii**

Chapter I. Introduction 1

1.1 Why Does Silicon Fail? 2
1.1.1 The Bathtub Curve 2
1.1.2 Silicon Failure Mechanisms 4
1.2 Defect-Tolerant Microarchitectures 7
1.3 The Reliable System Design Space 8
1.4 Contributions of This Thesis 10
1.5 Thesis Outline 12

Chapter II. Traditional Techniques and Recent Research Approaches for Defect-Tolerant Design 14

2.1 Traditional Defect-Tolerance Techniques 14
2.2 Fault Avoidance Strategies 16
2.3 Defect-Tolerance Techniques in Research Literature 17

Chapter III. Defect Tolerance Through Periodic Hardware Checking - The BulletProof Pipeline 21

3.1 Online Periodic Hardware Checking 22
3.1.1 Online Hardware Testing Techniques 24
3.1.2 Microarchitectural Checkpointing 28
3.1.3 Checkpointing with Two-Phase Commit 30
3.1.4 System Fault Recovery 31

3.1.5 Repairing the BulletProof Pipeline 31
3.1.6 Handling Input/Output Requests 32
3.1.7 Assumptions and Limitations 33
3.2 BulletProof Protection from Transient Faults 33
3.3 Experimental Evaluation 36
3.3.1 Experimental Framework 36
3.3.2 Testing Performance and Design Coverage 38
3.3.3 Run-time Performance 39
3.4 Related Work 42
3.5 Chapter Summary 45

Chapter IV. A Software-Based Periodic Hardware Checking Solution - The ACE Framework 47

4.1 Software-Based Periodic Defect Detection and Diagnosis 49
4.1.1 An ACE-Enhanced Architecture 49
4.1.2 ACE-Based Online Testing 53
4.1.3 ACE Testing in a Checkpointing and Recovery Environment . 56
4.1.4 Putting it Together: Algorithmic Flow of ACE-Based Testing . 56
4.1.5 ACE Testing Execution Models 57
4.1.6 Flexibility of ACE Testing 59
4.2 Experimental Methodology 60
4.3 Experimental Evaluation 63
4.3.1 Basic Core Functional Testing 63
4.3.2 ACE Testing Latency, Coverage, and Storage Requirements . . 65
4.3.3 Full-Chip Distributed Testing 67
4.3.4 Memory Logging in Coarse-grained Checkpointing 68
4.3.5 Performance Overhead of ACE Testing 69
4.3.6 Performance-Reliability Trade-off 73
4.3.7 Overhead of ACE Testing in I/O-intensive Applications 75
4.3.8 ACE Tree Implementation and Area Overhead 76
4.3.9 Power Consumption Overhead of the ACE Framework 78
4.4 Related Work 80
4.5 Chapter Summary 85

Chapter V. ACE Framework Extensions - Adding Value to Resiliency Mechanisms 87

5.1 ACE Framework for Online Design Bug Detection 88
5.1.1 The Problem of Design Bugs in Modern Microprocessors . . . 88
5.1.2 Design Bug Analysis 91
5.1.3 Detecting Logic Design Bugs at Runtime 98
5.1.4 ACE-Based Distributed Online Bug Detection 102
5.1.5 ACE-Based Segment Checking Tree Implementation 110

5.1.6 Experimental Evaluation 113
5.2 Related Work 118
5.3 Other Applications of the ACE Framework 122
5.3.1 ACE Framework Extensions for Post-silicon Debugging . . . 122
5.3.2 ACE Framework Extensions for Manufacturing Testing 128
5.4 Chapter Summary 130

Chapter VI. FPGA-Based Accelerated Hardware Resiliency Analysis - The CrashTest Framework 132

6.1 The Challenges of Hardware Resiliency Analysis 133
6.2 Overview of the CrashTest Framework 134
6.3 Gate-Level Fault Injection Methodology 136
6.4 FPGA-Based Fault Emulation 140
6.5 Framework Evaluation 141
6.5.1 Experimental Methodology 141
6.5.2 Monte Carlo Simulation & Statistical Confidence 143
6.5.3 Framework Performance 143
6.5.4 Experimental Results 145
6.6 Related Work 148
6.7 Chapter Summary 149

Chapter VII. Conclusions and Future Work 150

7.1 Thesis Summary 151
7.2 Thesis Conclusions 153
7.3 Future Work 156

BIBLIOGRAPHY **158**

LIST OF FIGURES

Figure

1.1 The Cost of Silicon Reliability: 2

1.2 The Bathtub Curve: 3

1.3 Reliable System Design Space: 9

2.1 Traditional Defect-Tolerance Techniques: 15

3.1 High-level System Architecture of the BulletProof Microprocessor: . . . 23

3.2 Component-Specific Online Hardware Testing Techniques: 25

3.3 Control Logic Checker Network: 27

3.4 Microarchitectural Checkpointing System: 29

3.5 Incorrect System Recovery Scenario: 30

3.6 BulletProof SER-Tolerant Flip-Flop Design: 34

3.7 Timing Diagram of a Transient Fault Detection: 35

3.8 The BulletProof Baseline Processor: 37

3.9 Performance Degradation of a Reconfigured BulletProof Processor: . . 41

4.1 ACE Framework Overview: 48

4.2 A Typical Scan Flip-Flop: 50

4.3 The ACE Architecture: 52

4.4 ACE Firmware: 54

4.5 Different ACE Testing Execution Models: 58

4.6 ACE Coverage of the OpenSPARC T1 Processor: 61

4.7 Fault Coverage of Basic Core Functional Testing: 64

4.8 Memory Logging Storage Requirements: 68

4.9 Performance Overhead of Single-Threaded Sequential ACE Testing: . . 69

4.10 Performance Overhead of SMT-Based ACE Testing: 71

4.11 Throughput Reduction Due to SMT-Based ACE Testing: 72

4.12 Performance Overhead of Interleaved ACE Testing: 73

4.13 Performance Overhead of ACE Testing VS. Test Coverage: 74

4.14 ACE Testing on I/O-Intensive Applications: 76

4.15 ACE Tree Implementation: 77

4.16 Power Consumption Overhead of the ACE Framework: 79

5.1 Design Bugs in Modern Microprocessors: 89

5.2 Overview of Online Design Bug Detection and Avoidance: 90

5.3 Design Bugs Documented in Microprocessor Errata Sheets: 92

5.4 Logic Design Bug: 94

5.5 Algorithmic Design Bug: 94

5.6 Timing Design Bug: 95

5.7 Design Bugs in the OpenSPARC T1 Core: 96

5.8 Design Bug Distribution: 97

5.9 Design Bug Triggering and Source Signals: 98

5.10 Design Bugs Source Signal Statistics: 99

5.11 Overview of ACE-Based Online Design Bug Detection: 103

5.12 Bug Detection Flip-Flop: 105

5.13 Bug Detection Example: 106

5.14 Bug Signature Merging: 108

5.15 Performance/Coverage Trade-off Tuning Algorithm: 109

5.16 ACE-Based Distributed Bug Detection: 110

5.17 ACE Framework for Online Design Bug and Defect Detection: 112

5.18 Area Overhead Versus Design Bug Coverage: 116

5.19 Power Consumption Overhead: 117

5.20 ACE Firmware for Post-Silicon Debugging: 127

6.1 Overview of the CrashTest Hardware Resiliency Analysis Framework: . 135

6.2 Logic Transformations - Bridge Fault: 138

6.3 Fault Injection Scan Chain: 138

6.4 Logic Transformation for the Path-Delay Fault Model: 139

6.5 FPGA-Based Fault Injection and Simulation: 140

6.6 Design Resiliency vs. Underlying Fault Model: 146

6.7 Failure Detection Latency: 147

6.8 Application-Level Detection Latency: 147

LIST OF TABLES

Table

3.1 Online Test Vectors: 38

3.2 Area Overhead of the BulletProof Technique: 39

3.3 Epoch Statistics for the Baseline Configuration: 40

3.4 Comparing BulletProof To Related Work: 43

4.1 The ACE Instruction Set Extensions: 51

4.2 Algorithmic Flow of ACE-Based Testing: 57

4.3 Test Instructions Needed to Test Each Major Modules: 65

4.4 Storage Requirements for Test Patterns and Responses: 66

4.5 Full-Chip Distributed ACE Testing: 67

4.6 Performance and Memory Log Size Tradeoffs: 70

4.7 Comparing The ACE Framework To Related Work: 82

5.1 Logic Design Bug Statistics: 100

5.2 Power Consumption Estimation Methodology: 114

5.3 Data and Control Signals in OpenSPARC T1: 115

5.4 Overhead of the Extended ACE Framework: 117

5.5 ACE Instruction Extensions for Post-Silicon Debugging: 125

6.1 Benchmark Designs: 142

6.2 Statistical Confidence: 143

6.3 Fault Injection Logic Overhead: 144

6.4 Fault Simulation Speed: 145

CHAPTER I

Introduction

For the last four decades, the semiconductor industry followed a trend known as the Moore's law [92]. Specifically, the Moore's law states that about every two years the transistor density of integrated circuits doubles. This means that about every two years, a microprocessor can have double the number of transistors in the same chip area. Since the release of the first commercial silicon-based microprocessor, almost forty years ago, the semiconductor industry was able to follow Moore's law due to the continued scaling of the silicon process technology that enables the fabrication of transistors with smaller dimensions. The major benefit of following the Moore's law is that with each scaling into a new silicon technology, every couple of years, the computer architects are offered smaller, faster, and cheaper transistors that makes possible the development of high-performance modern microprocessors. This technological achievement, over the last few decades, fueled the widespread adoption of microprocessor-based products in applications that touch every aspect of our life.

Currently, most mainstream consumer electronic devices are being produced with 65 and 45 nm silicon technology processes (that is the size of the smaller dimension in a transistor), and most microprocessor vendors are moving towards the adoption of the 32 nm silicon process technology. However, challenges in producing reliable components in these extremely dense technologies are growing, with many device experts warning that continued scaling will inevitably result in silicon technology generations that are much less reliable than the current ones [15, 123].

The cost due to the reliability challenges of future silicon process technologies is qualitatively illustrated in Figure 1.1. As shown in the graph, the primary benefit of technology scaling is the reduction in the cost per transistor with each new technology generation [44]. This trend makes the transition to newer technology generations more profitable to microprocessor vendors and it also enables the development of higher-performance microprocessors with more transistors. However, as the inherent reliability of new silicon

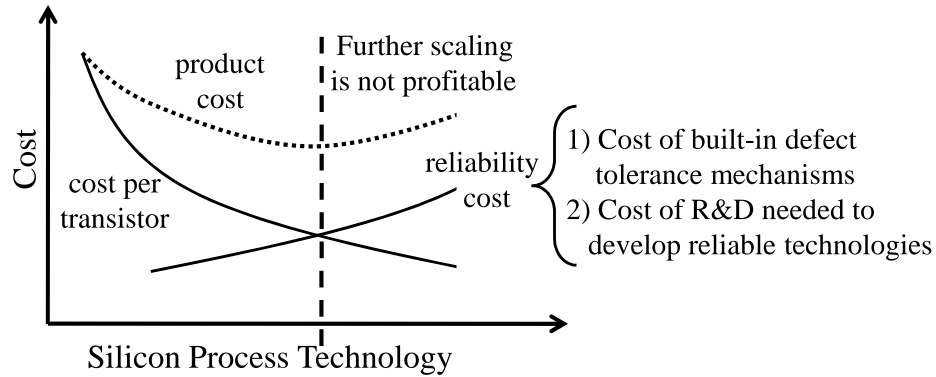


Figure 1.1: The Cost of Silicon Reliability: The graph shows the cost per transistor and the reliability cost for producing reliable microprocessors as the silicon process technology scales into future less reliable generations.

technologies wanes, we observe an increase in the reliability cost. The increase of the reliability cost can be due to either (i) the cost of shielding the microprocessors with built-in defect-tolerance techniques, or (ii) the cost of research and development (R&D) needed to develop new silicon process technologies that would allow the scaling to smaller feature sizes, but maintain the device reliability characteristics of the previous silicon process technologies. This reliability cost is contributing to the projected overall product cost. Experts warn that if this trend continues, eventually the silicon process technology scaling will reach a point where the reliability cost will overtake any benefits offered by smaller/cheaper transistors and any further scaling will be unprofitable for microprocessor manufacturing companies. This point is the minimum on the projected product cost curve shown in Figure 1.1.

To postpone or even eliminate this technology advancement barrier, the rate at which the reliability cost is increasing must be constrained. Technology experts suggest that this can be achieved by (i) building silicon-based semiconductor products out of unreliable components/technologies, and (ii) providing reliability to these products through online very low cost defect-tolerance techniques [17, 4]. The goal of this thesis is the exploration and evaluation of new, alternative, low-cost defect-tolerance solutions for microprocessor designs that will reduce the reliability cost induced by scaling into smaller and more unreliable silicon process technologies.

1.1 Why Does Silicon Fail?

1.1.1 The Bathtub Curve

Since the dawn of silicon processing, it has been recognized that the failure probability distribution function of silicon-based semiconductor electronic products over time is

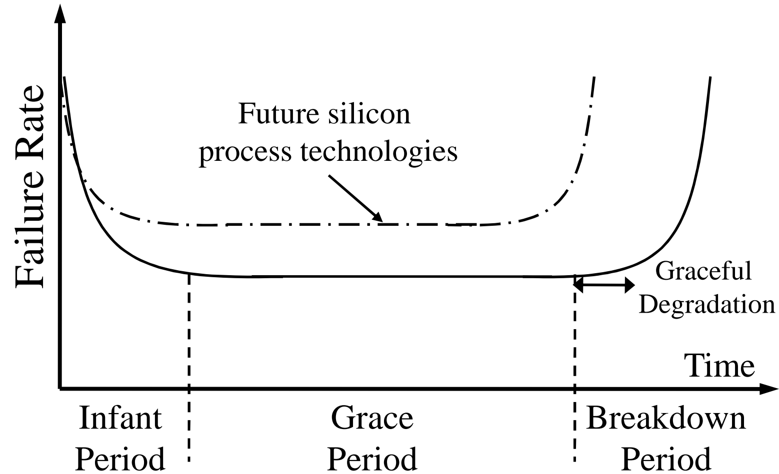


Figure 1.2: The Bathtub Curve: The bathtub curve indicates the qualitative trend of device failure rates for the population of a silicon-based semiconductor electronic product over time. The initial operational phase and the "aged-silicon" phase are characterized by much higher failure rates. The bathtub curves of future silicon process technologies are expected to shrink and exhibit higher device failure rates.

shaped like a bathtub. The bathtub-curve failure probability distribution function is characterized by three distinct phases as illustrated in Figure 1.2.

- **Infant Period:** The beginning of the product's lifetime is characterized by an initial high rate of device failures. These high failure rates are due to latent manufacturing defects that escape the initial product testing. These failures surface quickly when the manufacture-impaired devices are stressed as the products get into operation. However, the initial high failure rate declines rapidly as the remaining devices that pass the initial operating stress are more robust and less likely to fail.

- **Grace Period:** When early device failures are eliminated, the failure rate falls to a constant value where device failures occur sporadically due to the occasional breakdown of weak transistors or interconnect. It is highly desirable that the grace period will dominate a product's lifetime since this is the period where the product exhibits the lowest failure rates and the highest reliability.

- **Breakdown Period:** After the grace period, device failures start to occur with increasing frequency over time due to age-related wearout. Many devices will enter this phase at roughly the same time, creating an avalanche effect and a quick rise in device failure rates. However, since not all devices will fail at once, it is likely that a short graceful degradation period exists over which a few initial device failures begin to signal the onset of the device breakdown period.

As the silicon process technology scales into smaller transistor feature sizes, the bathtub curve of electronic products fabricated with these silicon process technologies is expected to shrink and exhibit higher failure rates. This will lead to products with shorter expected lifetimes. Furthermore, during their grace period, these products would be characterized by more frequent device failures.

The low-cost defect-tolerance solutions explored in this thesis are addressing the device failures that occur in the first two phases of the bathtub curve, namely, the infant period and the grace period. The objective of these mechanisms is to protect the microprocessor from occasional device failures that might occur early in its lifetime and tolerate the first device failures through the graceful degradation period. This strategy, offers to the user a time window to replace the defective part before the final breakdown.

1.1.2 Silicon Failure Mechanisms

Throughout the lifetime of a silicon-based semiconductor electronic product, its silicon fabric is subject to a variety of failure mechanisms that can cause device failures (leading to the previously mentioned bathtub curve). As the transistor dimensions scale to smaller sizes, these silicon failure mechanisms get aggravated. The following discussion highlights the types of device failures that are expected to characterize future silicon technologies. Each of these failure mechanisms has received significant attention in the silicon process technology literature, and each has been identified as a growing concern for deep-submicron silicon technologies. The interested reader can refer to [32, 103, 115, 125, 57] for a detailed treatment of these mechanisms.

Transistor Infant Mortality: Extreme device scaling exacerbates early transistor failures. Early transistor failures are caused by weak transistors that escape post-manufacturing validation tests. These weak transistors work initially, but they have dimensional and doping deficiencies that subject them to much higher stress than robust transistors. Quickly (within days to months) they will break down from stress and render the device unusable. Traditionally, early transistor failures have been reduced through aggressive burn-in testing, where, before being placed in the field, devices are subjected to high voltage and temperature testing, to accelerate the failure of weak transistors [23]. Those that survive the burn-in testing are likely to be robust devices, thereby ensuring a long product lifetime. However, in the deep-submicron silicon technologies, burn-in becomes less effective as devices are subject to thermal run-away effects, where increased temperature leads to increased leakage current, which in turn leads to even higher temperatures [87]. The end result is that aggressive burn-in of deep submicron silicon can destroy even robust devices. Manufacturers are forced to either sacrifice yield by deploying aggressive burn-in testing,

or experience more frequent early failures in the field by using less aggressive burn-in testing.

Manufacturing Defects that Escape Testing: Optical proximity effects, airborne impurities, and processing material defects can all lead to the manufacturing of faulty transistors and interconnect [103]. Moreover, deep-submicron gate oxides have become so thin that manufacturing variation can lead to currents penetrating the gate, rendering it unusable [115]. Even small amounts of manufacturing variation in the gate oxide could render the device unusable. The manufacturing defect problem is compounded by the immense complexity of current microprocessor designs. Design complexity makes it more difficult to test for defects during manufacturing. Vendors are forced to either spend more time with parts on the tester, which reduces profits by increasing time-to-market, or risk the possibility of untested defects escaping to the field. Moreover, in highly complex microprocessor designs, many defects are not testable without additional hardware support. As a result, even in today's manufacturing environment, untestable defects can escape testing and manifest themselves later on in the field during operation. All these problems are expected to worsen for future technologies and designs with smaller transistor feature sizes.

Time-Dependent Wearout: Technology scaling has adverse effects on the lifetime of transistor devices and interconnect, due to time-dependent wearout. There are three major failure modes for time-dependent wearout:

- *Electromigration*: Due to the momentum transfer between the current-carrying electrons and the host metal lattice, ions in the conductor can move in the direction of the electron current. This ion movement is called electromigration [32]. Gradually, this ion movement can cause clustered vacancies that can grow into voids. These voids can eventually grow until they block the current flow in the conductor. This leads to increased resistance and propagation delay, which in turn leads to possible device failure. Other effects of electromigration are fractures and shorts in the interconnect. The trend of increasing current densities in future technologies increases the severity of electromigration, leading to a higher probability of observing open and short-circuit nodes over time [41].

- *Gate Oxide Wear-out*: Thin gate oxides lead to additional failure modes as devices become subject to gate oxide wear-out (*e.g.*, Time Dependent Dielectric Breakdown, TDDB) [32]. Over time, gate oxides can break down and become conductive. If enough material in the gate breaks down, a conduction path can form from the transistor gate to the substrate, essentially shorting the transistor and rendering it useless

[41, 57]. Fast clocks, high temperatures, and voltage scaling limitations are well-established architectural trends that aggravate this failure mode [125].

- *Hot Carrier Degradation (HCD)*: As carriers move along the channel of a MOSFET and experience impact ionization near the drain end of the device, it is possible to gain sufficient kinetic energy to be injected into the gate oxide [32]. This phenomenon is called Hot Carrier Injection. Hot carriers can degrade the gate dielectric, causing shifts in threshold voltage and eventually device failure. HCD is predicted to worsen for future thinner oxide and shorter channel lengths [57].

Single-Event Upsets (SEU): There is also a growing concern about providing protection from single-event upsets (also known as transient errors or soft errors) caused by charged particles, such as neutrons and alpha particles, that strike the bulk silicon portion of a die [151]. Although SEUs do not break the silicon, their effect is a logic glitch that can potentially corrupt combinational logic computation or state bits. While a variety of studies have been performed that demonstrate the unlikeliness of such events [144, 142], concerns remain in the architecture and circuit communities. This concern is fueled by the trends of reduced supply voltage and increased transistor budgets, both of which exacerbate a design's vulnerability to SEU.

Process Variation: Another reliability challenge designers are expected to face in future silicon technologies is the design uncertainty that is created by increasing process variations. Process variations result from device dimension and doping concentration variation that occur during silicon fabrication. These variations are of particular concern because their effects on devices are amplified as device dimensions shrink [104], resulting in structurally weak and poor performing devices. Designers are forced to deal with these variations by assuming worst-case device characteristics (usually, a 3-sigma variation from typical conditions), which leads to overly conservative designs.

In many systems today, these silicon failure mechanisms are assessed, and the necessary margins and guards are placed into the design to ensure it will meet the intended level of reliability, essentially employing a fault-avoidance design strategy. For example, most transistor failures (*e.g.*, gate-oxide breakdown) can be reduced by limiting voltage, temperature, and frequency [59]. While these approaches have served manufacturers well for many technology generations, many device experts agree that silicon reliability will begin to wane as silicon processing scales in deep-submicron technologies. As devices become subject to extreme process variation, particle-induced transient errors, and transistor wearout, it will likely no longer be possible to avoid these faults. Instead, computer designers will have to begin to directly address system reliability through fault-tolerant design techniques.

1.2 Defect-Tolerant Microarchitectures

To address the concerns of silicon reliability, in this thesis we turn toward the development and application of defect-tolerant microarchitectures. In addition to their base functionality, defect-tolerant microarchitectures must support extra capabilities that will let the microprocessor to continue providing its intended service under the presence of silicon defects. A defect-tolerant microarchitecture, once it becomes aware of a defective part in the design, it needs to invoke a process that will reconfigure and repair the underlying hardware. After a silicon defect manifestation, the system also needs to be recovered from the defect's effects, including the restoration of any corrupted data or machine state. In order to address wearout-related silicon defects, or manufacturing defects that escape manufacturing testing, these capabilities need to be provided online while the product is operating in the field. Online defect tolerance is usually divided into the following four basic phases:

- **Error Detection:** Error detection is a vital capability for a defect-tolerant microarchitecture. Without error detection the system is unaware of the presence of any defects in the design and can lead to incorrect functionality that violates the system's specifications. Error detection can be accomplished by redundant computation, by error detection codes (*i.e.*, parity and error correction codes), or by checking the hardware for correct functionality. Unlike soft error detection, silicon defect detection through redundant computation requires the computation to be done on different hardware to avoid common mode failures.

  Error detection can be performed either at the macro-level or at the micro-level. Macro-level error detection is usually applied at the microprocessor scope by techniques like dual-modular redundancy and execution lock-stepping [118] that monitor the output of the microprocessor for errors. Micro-level error detection is usually applied at microprocessor subcomponents (*e.g.*, functional units, or the register file) by techniques like on-line built-in-self-test (BIST), residue checkers, or error detection codes.

  Since execution errors can be caused by both silicon defects and transients faults (due to neutron strikes, electrical noise *etc.*), error detection mechanisms are often required to distinguish the source of the error in order to invoke the necessary repair/recovery process. For example, restoring the processor's state and restarting execution is often adequate to recover from the effects of a transient fault. However, recovering from a permanent silicon defect is a more tedious process involving

defect diagnosis and hardware repair/reconfiguration. The ability of an error detection mechanism to accurately distinguish the source of an execution error is very important, since incorrect decision could disable or reconfigure functionally correct hardware resources that have been victims of transient faults and lead to impaired system functionality and/or performance.

- **Online Defect Diagnosis:** After an execution error detection, if a permanent silicon defect is indicated as the source of the error, an online defect diagnosis process is triggered in order to identify the defective component in the microprocessor design [20, 111]. During this process, the system needs to stall execution making online defect diagnosis a performance-critical operation.

- **Hardware Repair & Reconfiguration:** The reconfiguration and repair of hardware resources is an essential phase of defect tolerance for repairing a defective microprocessor and enabling the proper system functionality. Hardware repair can be handled in many ways, including disabling, ignoring, or replacing the defective hardware component. When there is enough hardware resource redundancy in the system, the hardware repair process can exploit this resource redundancy and simply disable the defective component. Alternatively, if there is enough redundant computation in the system (like there is in systems that employ triple-modular redundancy) the hardware repair may just be the ignorance of the defective component. Finally, when the system employs hardware sparing, the repair process replaces the defective component with a spare one.

- **System Recovery:** The final phase of online defect tolerance is the system recovery. After hardware repair, the system needs to restore any data and machine state that possibly got corrupted by the failure. System recovery essentially makes the manifestation of a silicon defect or a transient fault transparent to the application execution and provides correct system functionality to the user.

1.3 The Reliable System Design Space

When designing a defect-tolerant microarchitecture, there are two important design factors that need to be taken into consideration. The first, is the type of device failures that will be covered by the defect-tolerant microarchitecture. As discussed in the previous section, the types of device failures range from transient faults (SEUs) due to energetic particle strikes [151] and electrical noise [138], to permanent silicon wearout-related defects caused by electro-migration [50], stress-migration [59], and dielectric breakdown [147].

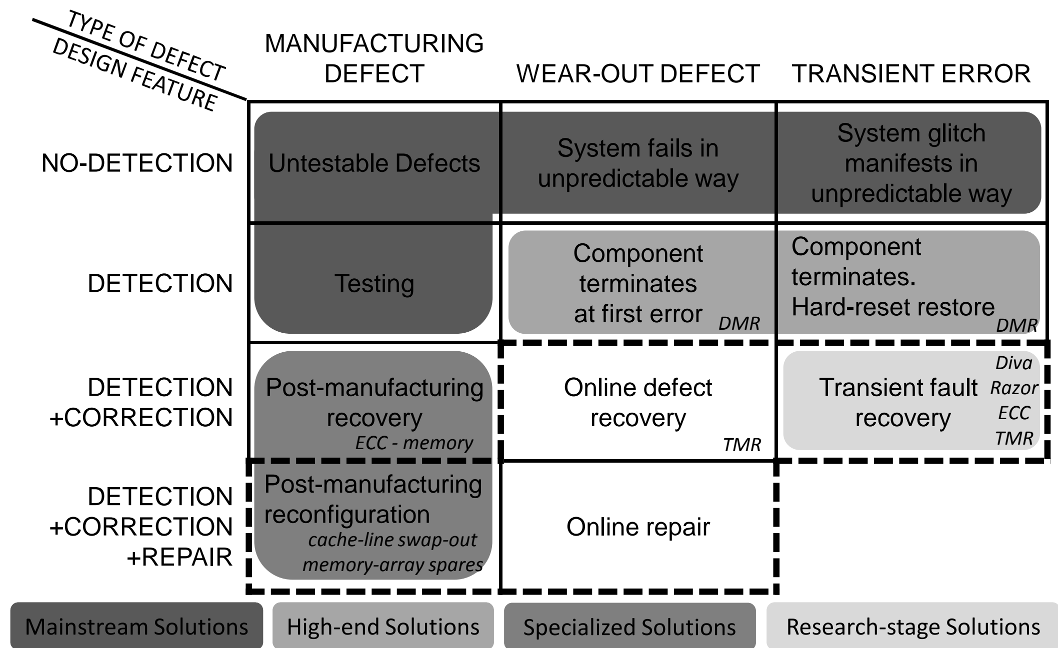


Figure 1.3: Reliable System Design Space: The diagram shows a map of device-level fault types in a digital system (horizontal axis) vs. protection techniques against these faults (vertical axis). This thesis addresses the problems/solutions in the dash bordered area of the map.

The second design consideration, is the degree to which the system will be protected from those device failures. Design solutions range from ignoring any possible device failures (as is done in many systems today), to detecting and reporting device failures, to detecting and correcting device failures, and finally failure correction with repair capabilities. This results to a rich design space to be considered, as illustrated in Figure 1.3. Specifically, Figure 1.3 illustrates the current fault-tolerant design space with the horizontal axis listing the type of device failure that systems might experience and the vertical axis listing the design solutions to deal with these device failures. Note that in this design space, the final two design solutions are the only solutions that can address permanent silicon defects, with the final solution being the only approach that maintains efficient operation after encountering a silicon defect.

In recent years, industry designers and academics have paid significant attention to building resistance to transient faults into their designs. A number of recent publications have suggested that transient faults, due to energetic particles in particular, will grow in future technologies [15, 93]. A variety of techniques have emerged to provide a capability to detect and correct these type of faults in storage, including parity or error correction codes (ECC) [117], and logic, including dual or triple-modular spatial redundancy [117] or time-redundant computation [31, 118] or checkers [68, 143].

In contrast, little attention has been paid into incorporating design tolerance for permanent silicon defects, such as transistor and interconnect wearout. The typical approach used today is to reduce the likelihood of encountering permanent silicon defects through post-manufacturing burn-in, a process that accelerates the aging process as devices are subjected to elevated temperature and voltage [147]. The burn-in process accelerates the failure of weak transistors, ensuring that, after burn-in, devices still working are composed of robust transistors. Additionally, many computer vendors provide the ability to repair faulty memory and cache cells, via the inclusion of spare storage cells [121]. Recently, academics have begun to extend these techniques to support sparing for additional on-chip memory resources such as branch predictors [19] and registers [114].

Currently, in the reliable system design space there are no low-cost defect-tolerance techniques that can provide effective mechanisms to online protect a microprocessor design from silicon defects, either those that occur during manufacturing or those that occur when the device is in operation in the field. This thesis will attempt to bridge this gap in the reliable system design space and explore defect-tolerance solutions that would cover the dash bordered area of the reliable system design space map of Figure 1.3.

1.4 Contributions of This Thesis

Traditional approaches to defect-tolerant design saddle a system with costly redundant components that continuously verify the integrity of all computation. Examples of such techniques are Dual Modular Redundancy (DMR) [117], and lockstep systems [64]. These techniques detect silicon defects by validating the execution through independent redundant computation. However, independent redundant computation requires significant hardware cost in terms of silicon area (100% extra hardware in the case of DMR and lockstep systems). Furthermore, continuous checking consumes significant energy and requires part of the microprocessor's power budget to be dedicated to it.

A major contribution of this thesis is a paradigm shift in the way that silicon defects can be detected in defect-tolerant microarchitectures. Rather than *continuously* checking computation for execution errors, the new approach is *periodically* checking the integrity of the underlying hardware without the need of redundant execution. This periodic hardware checking can be done through area-frugal, distributed, online checkers. This new defect-tolerance paradigm is relying on checkpointing and recovery mechanisms that provide computational epochs and a substrate for speculative unchecked execution. At the end of each computational epoch, the hardware is checked by on-chip testers. If the hardware tests succeed, the results produced during the epoch are committed and execution proceeds

to the next computational epoch. Otherwise, the system is deemed defective and system repair and recovery are required. A detailed prototype implementation of this approach, called *BulletProof*, is described and evaluated in Chapter III.

Another key requirement for a successful defect-tolerance solution is to have an ultra-low overhead in terms of silicon area, thus driving the overall product reliability cost low. Even though periodic hardware checking eliminates the high area cost of hardware replication required for redundant computation, it still requires a way to periodically check the underlying hardware. This could mean the addition of on-chip checkers and the bearing of their extra hardware cost, as we will observe later in Chapter III in the BulletProof prototype.

The amount of adaptivity and flexibility that defect-tolerance solutions provide, once the microprocessor is shipped and operating at the customer side, is also a central concern in their design. Flexible defect-tolerance solutions that can be modified, upgraded, and tuned in the field are very desirable. Today, many defect-tolerance techniques bind specific testing approaches into silicon, making it impossible to change the testing strategy after the microprocessor in deployed in the field.

To address both of these requirements and (i) offer low-overhead hardware checkers for periodic checking, and (ii) provide a flexible defect-tolerance mechanism that can be modified, adapted, and tuned to the needs of the microprocessor while it is operating in the field, we developed a new software-based defect-tolerance approach. The novelty of this new defect-tolerance approach is that it shifts the silicon defect detection and diagnosis process from on-chip hardware checkers to software. In this software-based defect-tolerance technique, called the Access-Control Extension (ACE) framework, the hardware provides the necessary substrate to facilitate hardware checking and the software makes use of this substrate to perform the hardware checking. The software nature of this approach offers a low area overhead mechanism for periodic hardware checking and inherently provides a flexible way for upgrading, modifying, and tuning the mechanism in the field. The ACE framework is described and evaluated in detail in Chapter IV.

Another challenge in the domain of defect tolerance for microprocessor designs is to overcome the expense of defect-tolerance mechanisms, which is necessary before they can be deployed in commercial mainstream microprocessor designs. One solution to this challenge is to add value to the defect-tolerance mechanisms by utilizing their hardware resources for more than just defect tolerance. To this extend, this thesis makes the case that the hardware resources used to implement a defect-tolerance solution can also be utilized for other applications. Specifically, as it will be demonstrated in Chapter V, the hardware resources of the ACE framework can be extended to other important applications such

as online design bug detection, a post-silicon debugging tool, and improving hardware manufacturing testing. This approach, adds value to defect-tolerance solutions and it can ease their early adoption in future generation microprocessors.

The last major contribution of this thesis is the development of *CrashTest*, a high-fidelity hardware resiliency analysis infrastructure on an FPGA-based emulation platform. Hardware resiliency analysis tools are used to assess the threats and the reliability requirements of a hardware design. During this process, faults are injected in the design and their impact on the behavior of the design is analyzed. After the fault injection and analysis process, the hardware design can be characterized for its reliability standards. However, the accurate assessment of the robustness of a hardware design is not a trivial process. Accurately modeling the effects of low-level silicon failure mechanisms and monitoring their impact up to the software level places conflicting requirements to the resiliency analysis tools. On the one hand, if low-level detail models of the hardware design are used to faithfully model the silicon failure mechanisms, the simulation performance is very poor and it limits the fault analysis from observing the impact of faults at the software level. On the other hand, if high-level architectural models of the hardware design are used to improve the simulation performance of the tool, the fidelity of the tool is in jeopardy since the effects of silicon failure mechanisms cannot be accurately modeled in high-level architectural models. In Chapter VI, the CrashTest hardware resiliency analysis tool makes an attempt to solve this conundrum by performing fault injection campaigns at the gate-level and accelerating the fault analysis process using an FPGA-based hardware emulation platform to achieve both accuracy and performance.

1.5 Thesis Outline

The remainder of this thesis is organized as follows:

Chapter II gives an overview of previous work done in the area of defect-tolerant microarchitecture design. It first highlights the traditional defect-tolerance techniques, followed by a discussion of their shortcomings. Chapter II also covers recent related work presented in the research literature.

Chapter III presents the *BulletProof* pipeline, a microprocessor defect-tolerance solution that employs periodic hardware checking coupled with microarchitectural checkpointing to provide low-cost protection from silicon defects. A description of the BulletProof physical-level prototype is provided, as well as a coverage and performance analysis in the context of a low-cost embedded VLIW microprocessor.

Chapter IV introduces the Access-Control Extension (ACE) framework, a software-based technique for online low-cost defect detection and diagnosis. The ACE framework

effectively moves the hardware checking process from the hardware to the software level. The ACE framework is evaluated on a commercial chip-multiprocessor system and experimental results and analysis are presented.

Chapter V extends the ACE framework to other applications. Specifically, it demonstrates that the ACE framework hardware resources can be extended and used for online design bug detection, as a post-silicon debugging tool, and for improving hardware manufacturing testing. Chapter V also provides an RTL-level design bug analysis of a modern commercial microprocessor that motivates the potential benefit of extending the ACE framework into an online design bug analysis mechanism. The chapter concludes with the experimental evaluation of the extended ACE framework.

Chapter VI presents *CrashTest*, an FPGA-based hardware resiliency analysis framework. The chapter starts with a high level overview of the CrashTest framework. Then, the gate-level fault injection methodology employed by the framework is described. The chapter continuous with the details on how CrashTest is implemented using a commercial FPGA. The chapter concludes with the experimental evaluation of the framework's performance and its effectiveness.

Finally, Chapter VII gives conclusions and discusses directions for future work.

CHAPTER II

Traditional Techniques and Recent Research Approaches for Defect-Tolerant Design

From the early adoption of computer systems, reliability was, and still is, one of the most important requirements in the design of computer systems. Computer applications ranging from life-critical aviation/ground transportation systems and medical systems, to business-critical applications found in the financial sector, to mission-critical applications like outer space exploration programs, they all place high hardware reliability demands to the computing systems. Furthermore, in applications such as outer space exploration, the computing systems are expected to operate in adversary environments and conditions that are very different from the ones here at Earth, such as very high rates of neutron strikes that cause transient faults.

These reliability requirements and challenges throttled a high research interest in reliable computer system design. This chapter, provides a brief overview of the traditional defect-tolerance techniques employed in high-end computing systems today and discusses their shortcomings. It also highlights the related work that was recently published in the research literature.

The scope of this chapter is to present the previous work that is more relevant to the general topic of this thesis. Other previous work that is related with the specific techniques described in each of the remaining chapters of this thesis is presented in each chapter respectively.

2.1 Traditional Defect-Tolerance Techniques

One of the first defect-tolerance approaches used to protect high-end computer systems is dual modular redundancy. Dual modular redundancy, employs spatial redundancy in the form of two microprocessors operating in lockstep. The output of the two microprocessors

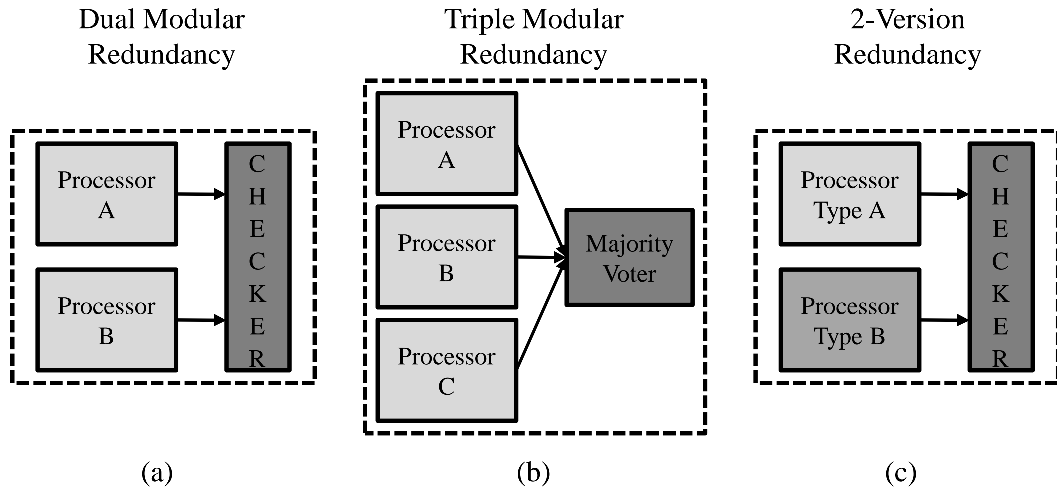


Figure 2.1: Traditional Defect-Tolerance Techniques: Part (a) shows a dual redundancy system where two identical processors are operating in lockstep and checked by an external checker. Part (b) shows a triple modular redundancy system where errors are both detected and corrected. Part (c) shows a 2-version redundancy system where two different processors, with the same specifications, are running in lockstep. This approach avoids common mode failures and also detects design bugs.

is checked by an external checker, as shown in Figure 2.1(a). 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 [64].

One shortcoming of dual modular redundancy is that although it can effectively detect single defects in the design, once a defect is detected, it cannot detect which of the two microprocessors is the defective one and continue operating with the other one. Therefore, once a defect 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 [117]. In triple modular redundancy, three identical microprocessors are used with an additional majority voter, as shown in Figure 2.1(b). 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 defect-free microprocessors.

Another similar approach to dual and triple modular redundancy is N-version redundancy. With N-version redundancy, instead of just replicating the microprocessors N times as in modular redundancy, N different microprocessors, with the same specifications, are designed by N different design teams or companies. An example of a 2-version redundancy system is shown in Figure 2.1(c). The N-version redundancy has the additional advantage over modular redundancy of detecting not only hardware failures, but also design bugs and avoiding common mode failures.

The advantage of these hardware redundancy techniques is that they are not intrusive in the microprocessor design and they can be applied to build defect-tolerant computer systems using over-the-shelf processors. Also, these techniques cover uniformly the whole microprocessor and can detect errors caused by any defective structure in the processor. However, the major shortcoming of these techniques is that they add extra hardware into the system leading to significant area and power overheads.

Another traditional defect-tolerance technique used to protect memories, buses, or other microprocessor array structures (*e.g.*, register file) are parity and error correction codes (ECC) [117]. 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. However, ECC and parity bits are intrusive in the design of the microprocessor[1] and protect only a limited part of the processor.

In the context of online testing of processors, various concurrent error detection schemes have been proposed [89]. Most schemes incorporate a checker that compares the expected behavior with that of the unit under test. Another solution proposed in the direction of online testing are Berger codes [12] which can detect all unidirectional errors, and Bose-Lin codes [16] which can detect t unidirectional errors (known as t-EC). These codes are suitable for the protection of circuits that are skewed towards one of the two logic values (logic 0 or 1). However, the use of these codes for online testing of datapaths is not trivial as they impose constraints on the way the logic block is designed such that only unidirectional faults occur. As with the parity and ECC bits, concurrent error detection schemes and the Berger codes are intrusive in the design of the microprocessor and protect only some parts of the processor design.

2.2 Fault Avoidance Strategies

Today, the defect-tolerance techniques presented in the previous section are only used in high-end systems running critical applications. In contrast, the microprocessors used in most mainstream desktop and laptop computers and embedded systems employ a fault-

[1] This means that they need changes at the design phase of the processor, unlike modular redundancy techniques that can be applied to over-the-shelf processors.

avoidance design strategy to achieve their projected failure rate targets. Microprocessor manufacturing companies assess the sources of possible failures, and they place in the design the necessary guards and preventive measures to ensure that their exposure to failing scenarios does not compromise the overall reliability target. The incident of silicon failure mechanism, such as wearout-related silicon defects and transient faults, is proportional to supply voltage, circuit temperature, and transistor activity factors [59], thus, reliability in these microprocessors is typically ensured through the use of safety margins inserted into the clock period and limits on the maximum supply voltage.

If the microprocessor failure rate resulting from ignoring the occurrence of the faults falls within the targeted reliability standards, the sole use of fault avoidance techniques is adequate to provide a relatively reliable population of products. Although previous and current generations of silicon process technologies exhibit sufficiently low device failure rates that silicon defect could be completely ignored, this approach is expected to be ineffective for future silicon technologies where device wear-out, untestable defects, and early transistor failures will increase the in-the-field (*i.e.*, during operation) microprocessor defect rates and necessitate stronger measures of protection.

2.3 Defect-Tolerance Techniques in Research Literature

To date, only a few efforts have explored techniques to provide low-cost defect tolerance to microprocessor designs. This section, provides a brief overview of the previous research work that is more generic to the subject of microprocessor defect tolerance. Previous research works that are more relevant to the specific techniques explored in the remaining chapters of this thesis are described and discussed in those chapters respectively. Also, some of the research work discussed in this section was concurrently developed with the work presented in this thesis.

Defect Tolerance Through Continuous Execution Checking: DIVA, is an online checker component inserted into the retirement stage of a microprocessor pipeline that continuously validates the computation, communication, and control exercised in a complex microprocessor core [6, 143]. 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.

To address this limitation, Bower *et al.*, in [20], propose a fault-tolerant microprocessor design that uses DIVA checkers for system-level error detection coupled with a mechanism

for diagnosing silicon defects by tracking the instruction occupancy through the microprocessor's pipeline. After diagnosing a silicon defect, the microprocessor reconfigures (*i.e.*, disables) the defective part and continues operation at a gracefully degraded level of performance.

More recently, Meixner *et al.* presented Argus [85], an error detection technique for simple processor cores. The Argus technique continuously checks invariants to detect execution errors. Specifically, Argus, uses run-time invariant checking to detect errors in four fundamental tasks: the control flow, the dataflow, computation, and memory access. The Argus technique provides error detection for errors caused by both permanent silicon defects and transient faults and offers an alternative low-cost defect-tolerance approach when compared to the traditional defect-tolerance approaches.

Hardware Testing and Built-In-Self-Test: After chip fabrication, microprocessor chips are tested in order to screen out parts with defective or weak devices. Today, most complex microprocessor designs use scan chains as the fundamental design for test (DFT) methodology. During hardware testing, the design's scan chains are driven by external automatic test equipment (ATE) that applies pre-generated test patterns to check the chip under test [23]. Every single microprocessor chip has to go through this testing process multiple times at different voltage, temperature, and frequency levels. This makes the manufacturing testing cost for each chip to be as high as 25-30% of the total microprocessor manufacturing cost [45]. An alternative approach that eliminates the need of external equipment to drive the hardware testing is Built-In-Self-Test (BIST) techniques [23]. BIST techniques use specialized circuitry to generate test patterns and validate the test responses on the chip without the need of any communication with external devices. The way BIST techniques generate test patterns on the chip is either by the use of pseudo-random test pattern generators, or by storing previously generated test vectors in on-chip memories.

Silicon Defect Prediction: Blome *et al.* [14], proposed an online technique that detects the performance degradation caused by wearout over time in order to anticipate failures. In particular, the proposed technique leverages the progression of wearout over time and provides a low-overhead self-calibrating hardware structure that identifies increasing propagation delay, which is symptomatic of many forms of wearout, to forecast the failure of microarchitectural structures. Specifically, they propose the implementation of an online latency sampling unit that is capable of sampling and filtering by statistical analysis the propagation latencies of signals to identify significant changes in the latency of a given microarchitectural structure and predict a device failure.

In [129], Sylvester *et al.* propose in the context of the ElastIC architecture the use of in-situ sensors in combination with reliability and power models to predict the lifetime

and wearout of the underlying hardware. This enables the dynamic tradeoff of performance with longer lifetime and reliability using dynamic voltage scaling techniques. A similar approach was employed by Srinivasan *et al.* in [124] where microarchitectural components are swapped by spare ones based on the prediction of their failure. The failure time of microarchitectural components is predicted by monitoring the dynamic activity and temperature of the microarchitectural components in combination with the analytical reliability models proposed in [122].

Resource Redundancy for Hardware Repair & Reconfiguration: Shivakumar *et al.* [114], proposed the use of hardware redundancy and reconfiguration to improve the yield and increase the defect tolerance of future microprocessors. They also suggest that the use of hardware redundancy should not be limited only to memories but 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).

In [42], Gupta *et al.* presented StageNet, a highly reconfigurable multicore architecture. StageNet is a reconfigurable multicore computing substrate designed as a network of pipeline stages, rather than isolated cores in a chip-multicore processor. The StageNet network is formed by replacing the direct connections at each pipeline stage boundary by a crossbar switch. Within the StageNet network, pipeline stages can be selected dynamically from the pool of available stages to form logical processing cores, thus permanent silicon failures can be easily isolated by adaptively routing around defective stages. In essence, the StageNet substrate can effectively exploit the natural resource redundancy of moder multicore processors and reconfigure the hardware resources around a defective component to repair a microprocessor design.

Aggarwal *et al.* [3] introduced the notion of configurable isolation for low-level fault containment and component reconfiguration through cost-effective modifications to commodity designs. 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. Furthermore, the power budget of the defective disabled components is re-assigned to the remaining operating components. This enables the voltage/frequency upscaling of the remaining hardware resources in an attempt to mitigate the performance loss due to the disabled components.

Bower *et al.* [19] proposed a hardware mechanism for self-repairing array structures to provide defect detection and repair capabilities for microprocessor array structures such as the reorder buffer and branch history table. The proposed mechanism detects silicon defects by employing dedicated "check rows". Every time an entry is written to the array structure, the same data is also written into a check row. Then, both locations are read out and their values are compared to detect defective rows. To repair defective arrays, the mechanism exploits the inherent resource redundancy of these structures and redirects any accesses to defective rows to other functionally correct rows.

Finally, an algorithmic approach for dynamic hardware reconfiguration and system repair from silicon defects was proposed recently by Fick *et al.* in [34]. Specifically, the work presented in [34] proposes a distributed routing algorithm for networks on chip that allows a network to reconfigure around defective components. The proposed algorithm is able to overcome large number of silicon defects by running in lockstep at each network router and collectively reconfiguring the network's routing tables. It was demonstrated that due to the high hardware resource redundancy of networks on chip, the dynamic reconfiguration algorithm could provide a 99.99% reliability to the on-chip network even after 10% of its interconnect links were defective.

CHAPTER III

Defect Tolerance Through Periodic Hardware Checking - The BulletProof Pipeline

This chapter introduces *BulletProof*, an ultra low-cost defect-tolerance mechanism to protect a microprocessor pipeline and on-chip memory system from permanent silicon defects. The traditional approach to defect tolerance saddles a system with redundant components that continuously monitor the microprocessor's execution for errors through redundant computation. This redundant computation leads to significant area and power overheads and constraints the microprocessor's resource budget. The BulletProof technique shifts the traditional defect-tolerance paradigm from continuous checking for execution errors (through redundant computation) to periodic online hardware checking. Specifically, it combines area-frugal periodic online hardware testing with microarchitectural checkpointing to provide the same guarantees of reliability as traditional defect-tolerance techniques, but at a much lower cost.

This approach, utilizes a microarchitectural checkpointing mechanism to create coarse-grained epochs of execution, during which a distributed online hardware testing mechanisms verify the integrity of the underlying hardware. If the hardware is deemed unbroken at the end of a computation epoch, the epoch's speculative computation is allowed to retire to a non-speculative system state, otherwise, the system is rolled back to the beginning of the epoch, and the last known-good system state is restored. At recovery, the system is reconfigured to disable any defective components. This technique relies on the natural resource redundancy that is abundant in ILP-style microprocessors combined with a small amount of carefully-placed control logic redundancy to repair the system such that it can operate in a degraded performance mode. Once repaired, the user can decide whether to replace the system or tolerate the degraded performance. The BulletProof technique also employs a double-sampling flip-flop design to protect the pipeline from transient faults and latch defects.

Section 3.1, describes in detail the implementation of the BulletProof defect-tolerance technique, including a detailed coverage of the distributed checkers used to perform the periodic hardware testing. It also explains how the technique employs microarchitectural checkpointing to provide speculative computational epochs and system recovery, how the BulletProof pipeline is repaired, and how input/output operations are handled by the BulletProof mechanism. Next, Section 3.2 describes in detail the double-sampling flip-flop design used to protect the BulletProof pipeline from transient faults. Section 3.3 evaluates the BulletProof mechanism using both detailed circuit-level and architectural simulation. The simulation testbed used for the evaluation of the BulletProof mechanism is based on a low-cost embedded VLIW processor. Finally, Section 3.4 discusses previous research work that is related to the BulletProof technique, and Section 3.5 summarizes the work presented in this chapter.

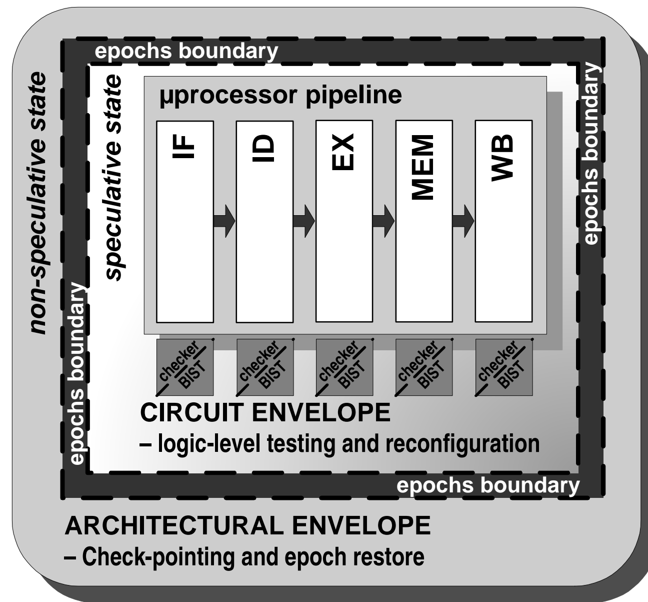
3.1 Online Periodic Hardware Checking

Figure 3.1 illustrates the high-level system architecture of the BulletProof defect-tolerance approach, and it shows a timeline of execution that demonstrates its operation. At the base of the proposed approach is a microarchitectural checkpoint and recovery mechanism that creates *computational epochs*. A computational epoch is a protected region of computation, typically at least 1000's of cycles in length, during which the creation of any errant computation, in this case due to the encountering of a defective device, can be undone by rolling the computation back to the beginning of the computational epoch.

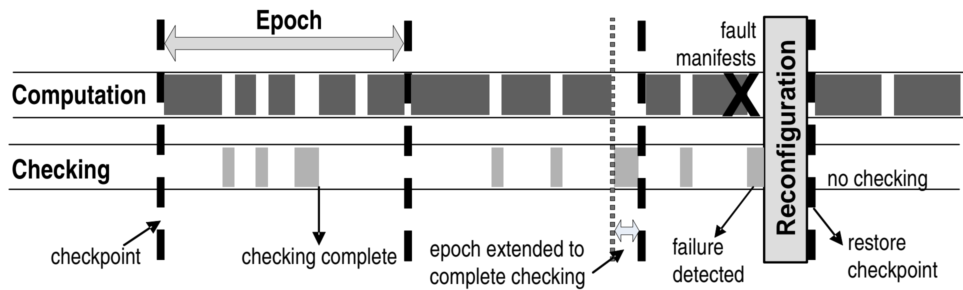
During a computational epoch, online checkers perform hardware built-in-self-test routines in the background, checking the integrity of all system hardware components. Ideally, this hardware checking will occur while functional units, decoders, and other microprocessor components are idle, as is often the case in a processor with parallel resources.

By the end of a computational epoch, there are three possible scenarios that the BulletProof mechanism will need to handle. The first scenario (shown in the first computational epoch of Figure 3.1(b) is when the checking completes before the end of the computational epoch. In this scenario, the hardware is known to be free of defects, thus, the results of the computational epoch are known to be free of defect-induced errors, and it can be safely retired to non-speculative system storage.

In the second scenario (shown in the second epoch of Figure 3.1(b), the computational epoch ends before the online testing infrastructure has completed testing all of the underlying hardware components. This scenario can occur because the microarchitectural



(a)



(b)

Figure 3.1: High-level System Architecture of the BulletProof Microprocessor: The schematic in part (a) shows an overview of the BulletProof protected microprocessor. Part (b) shows three possible execution scenarios within a computational epoch.

checkpointing mechanism has only a finite amount of storage into which speculative state can be stored – once this space is exhausted, the computational epoch must end. Additionally, I/O requests can force early termination of a computational epoch. In the event the computational epoch completes before testing is finished, testing will continue with the processor pipeline stalled. If at the end of testing the hardware is deemed free of defects, the epoch's speculative state can safely retire to non-speculative system storage.

Finally, the third scenario, depicted in the third epoch of Figure 3.1(b), is when the online testing infrastructure encounters a defect in an underlying component, due to a transistor wearout, early transistor failure, or manifestation of an untested manufacturing defect. In this event, the execution from the start of the computational epoch to the point where the defect was detected cannot be trusted as correct, because this unchecked computation may have used the defective component. Consequently, the results computed during the epoch

are discarded, and the underlying hardware must be repaired. This is done by disabling the defective component. In a processor with instruction-level parallelism (ILP), there are typically multiple copies of virtually all hardware components. Once a component is disabled the processor will continue to run in a performance-degraded mode. Additionally, a software interrupt is generated which notifies the system that the underlying hardware has been degraded, so the user can optionally replace the impaired processor.

3.1.1 Online Hardware Testing Techniques

The online hardware testing infrastructure is responsible for fully verifying the integrity of the underlying hardware components. The testing techniques are adopted from built-in self-test (BIST) [22], although they are tailored to minimize the area of the testing hardware, and hence the area of the defect-protection infrastructure. For each of the pipeline components, a high quality input vector set is stored in an on-chip ROM, which is fed into the modules during idle cycles. A checker is also associated with each component to detect any defect in the system. The primary techniques utilized to verify the integrity of the underlying hardware are illustrated in Figure 3.2 and described below.

Decoder Checker: The decoders are validated by sending the same test vector to multiple decoders, and then comparing their outputs. The decoder test harness is illustrated in Figure 3.2(a). In the event that the outputs do not match, one of the decoders has experienced a defect-related failure. In addition, it is important to determine which of the decoders has failed. Consequently, three decoders are sent the test vector, and a majority operator is used to identify which of the decoders has failed. In the case that the architecture has more than three decoders, each can be tested by including it in a battery of tests with any two other decoders.

Register File Checker: Register file integrity is checked using a four phase split-transaction test procedure, as illustrated in Figure 3.2(b). The register file is unchanged from the original design, except that it has two address decoders (one for read and one for write), which allows the testing of address decoder faults. In the first phase, a register file entry is read from the register file and stored in the *replacement register*. Testing of that register may now proceed whenever free read/write ports are available. If the register under test is read or written by the processor, the value is supplied by the replacement register. This same register is used to repair a broken entry, as described later. In the second phase, a random vector (generated with a linear feedback shift register, LFSR) is written into the register being tested, and in the third phase it is read back out and compared to the original vector. Finally, in the last phase the register entry (previously copied into the replacement register during the first phase) is written back into the appropriate register.

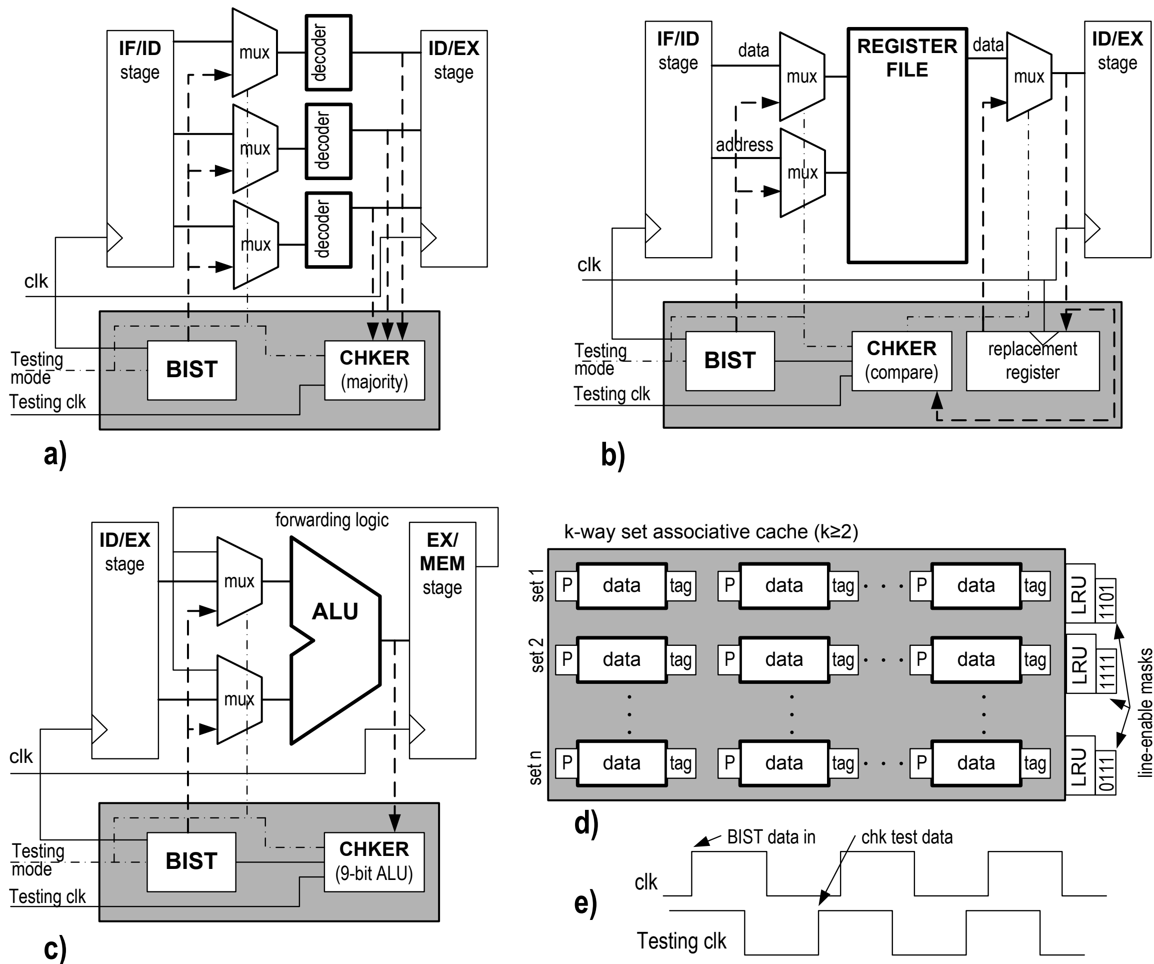


Figure 3.2: Component-Specific Online Hardware Testing Techniques: The decoders use a majority vote, as shown in part (a). In part (b) the register file tests one register at a time, by swapping to a replacement register. Part (c) shows functional units exploiting arithmetic checkers. In part (d) caches are equipped with a parity bit, a "volatile" bit to indicate the speculative state of the data stored in a line and bits to track a faulty cache line. Part (e) shows the early clock edge for checker logic.

This process effectively tests both the register storage as well as the address decoders in the register file. The register storage is tested by writing and reading a value from the register. The address decoders are tested by virtue of the fact that the value written and read is fairly unique (*i.e.*, it is randomly generated), thus if either the read or write address decoder incurs a defect, some other (likely another register value) value will incorrectly appear during the read phase of the register file testing. Because the value stored in the register entry under test is available at all times from the replacement register, the testing process can be implemented as a series of split transactions. Consequently, different phases can be executed in non-subsequent cycles, whenever a free port is available on the register file. This facet of the approach greatly contains the performance impact, as shown

in Section 3.3. The register file testing procedure is repeated until all of the registers have been validated. For a processor with 32 registers, the register file can be fully tested within 128 cycles, spread out over an entire computational epoch in cycles when the register file is not in use.

ALU and Multiplier Checker: The ALU is checked using a 9-bit mini-ALU, as shown in Figure 3.2(c). During each cycle a test vector from the BIST unit is given to the ALU and compared with the output of the mini-ALU. It takes four cycles for the mini-ALU to test the full output of the main ALU. A 9-bit ALU is used to validate the carry out of each 8th bit in the 32-bit output. The same type of ALU checker is also used to verify the output of the address generation logic. Using the mini-ALU checker, it is possible to fully verify that the ALU circuitry is free of stuck-at-0 and stuck-at-1 faults with only 20 carefully selected test vectors.[1] A similar approach is used to validate the multiplier, which employs arithmetic residue checks [7]. Given an n-bit operand x, the residue x_r with respect to r is the result of the operation $x\%r$. When applied to multiplication, residue codes adhere to the following property: $(x_r * y_r) = (x * y)_r$. When the value of $r = 2^a - 1$ for some a, the residue operations are much simpler to implement in hardware [7]. The resulting multiplication checker requires only a shifter and simple custom logic.

Residue codes can detect most of the faults in a multiplier except those that manifest as multiples of the residue, a small class of faults where a single fault at an internal node could manifest as a multiple of the correct value on the output. The errors missed by the residue checker are caught by a few additional carefully selected test vectors, against which the exact output is matched.

Cache Line Checker: Cache line integrity is maintained, as illustrated in Figure 3.2(d), through the use of cache line parity. A single parity bit is associated with each line, holding the parity of the entire cache line plus the tag, valid bit, and LRU state for the line. When cache lines are written to the cache, the parity for the line is generated and stored. Subsequently, when the cache line is read, the parity is recomputed to verify the contents. In the event that the parity is correct, notwithstanding a multi-bit failure, which is beyond the scope of the single bit failure model, the cache line is known to be correct. In the event that a cache line parity check fails, a defect has been detected within the storage of the cache, consequently, the line must be disabled from further use and execution is rolled back to the last checkpointed epoch. Cache lines are disabled by setting a two bit field in the LRU state table, which indicates which line in the current set has been disabled. The disable

[1]It should be noted that this testing approach is in contrast to traditional BIST-style testing techniques that store both the input and output vectors, with the output vectors being compared to the output of the ALU. By computing the output vector on a smaller adder, a tester that was significantly smaller could be produced.

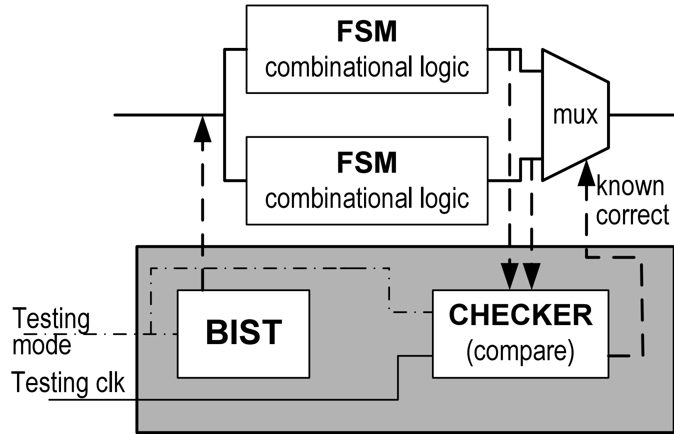


Figure 3.3: Control Logic Checker Network: The figure shows the checker mechanisms deployed for protecting the control logic in the microprocessor pipeline.

bits in the LRU table are periodically reset to avoid soft errors in caches being interpreted as hard errors and rendering the cache lines unusable for the rest of the design's lifetime. Furthermore, at the end of each computational epoch, dirty cache lines are checked and written back to the next level of the memory hierarchy to guarantee recoverability in the presence of cache silicon defects. This approach is area-efficient, but it can only support a single failed line per set of a cache. Additional failed lines could be supported within a single set if more disable bits were to be included in the LRU logic.

The Test Clock: An important consideration in the testing of hardware components is the timing of the test vector samples. Since many transistor wearout-related failures manifest as progressively slower devices [41], the failure of the device may occur in a way where timing is no longer met for the component's critical path. Figure 3.2(e) shows how this issue is addressed by utilizing a slightly shorter clock cycle for sampling test vector outputs. The clock frequency safety margins in current microprocessors (*e.g.*, to mitigate process variation) permits the use of this slightly shorter cycle testing clock with a negligible amount of false positives. This ensures that if a device is failing by showing slower response, it can be detected long before it affects any processor computation, which operates on the main clock cycle that is longer than the testing cycle.

Protecting Control Logic: To achieve high fault coverage it is critical to protect the control logic, since this logic constitutes a non-trivial fraction of the area in most microprocessor designs. For the protection of the pipeline's control logic, a dual-modular redundancy based approach is employed, as illustrated in Figure 3.3. Two copies of the pipeline control logic run in parallel, each with the same set of inputs. Every cycle, the outputs of the control blocks are compared and if any difference occurs, a fault is flagged. To localize the fault, built-in-self-test is used to determine which of the two control block

copies is defective. Once identified, the defective control logic block is permanently disabled. Note that this approach can only tolerate defects to the extent that they occur in only one of the two control logic blocks. This technique has area-cost advantages over triple-modular redundancy because checker and built-in-self-test logic are typically smaller than a third copy of control logic as required to implement triple-modular redundancy. Note also that each control logic block is protected individually, leading to smaller overhead in the interconnect and higher resiliency.

Checker, Check Thyself: The checkers constitute a non-trivial portion of the microprocessor's area.[2] Consequently, if the checkers themselves were not checked, they would severely limit overall design fault coverage. To keep area cost low, checkers are checked using the same component they monitor, a technique called *reflexive self-test*. In other words, the online checkers are designed such that they produce a correct result only when *both* the unit-under-test and the checker are free of silicon defects and other faults. For example, a built-in-self-test vector generator and a 9-bit adder is used to check the processor's adder. At the same time, the processor's adder is used to test the functional integrity of the built-in-self-test vector generator and the 9-bit adder.

In traditional testing the built-in-self-test vectors are selected so that they have a high probability to detect defects in the unit under test. In reflexive testing, there is an additional constraint that the test vectors must also expose defects in a broken checker (assuming that the unit under test is still working). Consequently, assuming a single-defect fault model, a built-in-self-test routine will fail if there is a defect either in the unit-under-test or in the checker. If the defect is in the checker, the end result will be the disabling of the working unit and its broken checker, hence the desired result of disabling the defective checker component is achieved as a byproduct.

3.1.2 Microarchitectural Checkpointing

The BulletProof technique relies on a microarchitectural rollback mechanism to restore correct program state in the event of a defect detection. The employed mechanism is similar to the one described in [80]. During the execution of a computational epoch, the processor generates register and memory updates which would need to be discarded if a fault is detected. To prevent any memory updates with corrupted data, such updates are buffered in speculative state within the processor, until when the hardware is checked and certified to be functionally correct. It is worth noting that the same level of fault coverage is not feasible by simply stopping the computation and running the built-in tests on a regular basis, without any checkpointing, and reconfiguring the pipeline if a fault is found. In fact,

[2] More than 10% of the area of our prototype design.

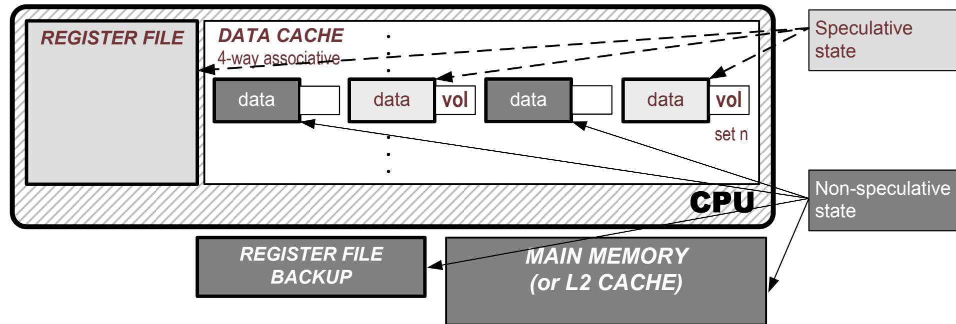


Figure 3.4: Microarchitectural Checkpointing System: The storage in dark color indicates all the state that is non-speculative at any execution point. Light colored storage is speculative during each epoch, and it is only synchronized at epoch boundaries.

with this approach it would not be possible to ensure that a detected fault had not corrupted earlier computation. In contrast, with the microarchitectural checkpointing facility, the state of the machine can always be rolled back to the point when the last completed online test passed successfully (a point in the computation known to be correct). In addition, once the hardware is repaired, the program can safely restart from this checkpoint.

As shown in Figure 3.4, register state is preserved by backing up the register file into a dedicated single-port SRAM at the beginning of each computational epoch. The register backup can be done lazily by tagging the registers and copying them only before they get overwritten, so that there is no associated performance penalty.

To support long epochs, memory updates are buffered within the local cache hierarchy. To implement in-cache speculative state, each cache line is augmented with a *volatile bit*. At the beginning of an epoch, all volatile bits are reset. When a value is stored to the cache, the volatile bit of the target cache line is set to indicate that the contents are speculative in the current epoch. The end of an epoch is determined by the ability of the local cache hierarchy to buffer the memory updates issued during the epoch. Therefore, the end of an epoch is triggered by a cache miss on a cache set with all its cache lines already been marked as volatile. In this event, all speculative state resources have been exhausted and the processor must stall until the testing sweep is complete. Once the underlying hardware is determined to be defect-free, an epoch may end. At this point, all volatile bits from the cache lines are cleared, changing all formerly speculative state to non-speculative state.

To minimize the performance cost of starting a new epoch (*i.e.*, copying the register file and clearing volatile bits), each epoch is extended as long as possible, until when speculative state resources are exhausted or a high-priority I/O request is generated, as discussed in Section 3.1.6. To provide even longer epochs, a small fully associative victim cache for volatile cache lines is introduced, so that the end of an epoch is now triggered by a cache miss on a cache set with all its lines been marked as volatile, and while the victim cache is

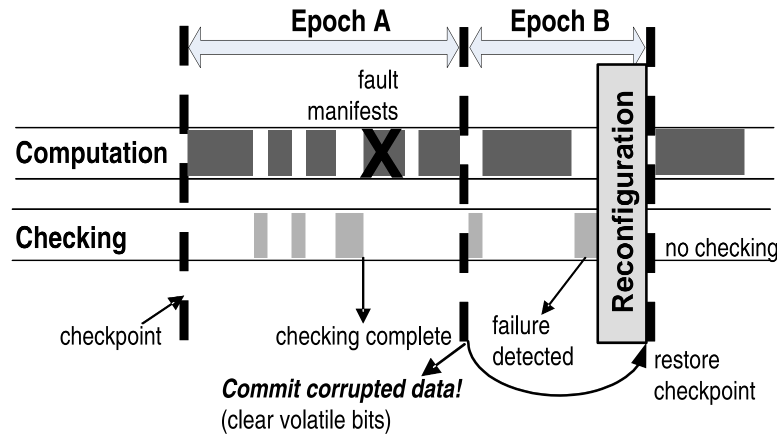


Figure 3.5: Incorrect System Recovery Scenario: During the execution of epoch A, a fault manifests after the testing sweep is complete. The fault causes memory updates with corrupted data, which are committed at the end of the epoch. In epoch B, the fault is detected and recovery occurs. However, this happens too late to revert the corrupted memory updates of epoch A.

full of volatile lines. This work assumes a uni-processor environment; hence, delaying the commit of stores to non-speculative storage has no effects on the system's performance. Similar microarchitectural checkpointing techniques that address the performance penalty of delayed stores in shared-memory multi-processor environments are described in [58].

3.1.3 Checkpointing with Two-Phase Commit

Unfortunately, if only one checkpoint of the microprocessor's architectural state is preserved, there is a chance that errant computation from a silicon defect could be missed. The potential problem is illustrated in Figure 3.5: If a hardware check completes before a fault manifests, it becomes possible for an errant computation to be generated *later* in the same computational epoch. In this event, corrupted state updates would be committed to non-speculative state at the end of the epoch. The manifested fault will eventually be detected in the next epoch, but not before erroneous computation had a chance to be committed to non-speculative storage. This conundrum can be solved by adopting a two-phase commit procedure, which maintains two checkpoints of the processor's state.

To implement this two-phase commit, an additional bit is used for each L1 data cache line. An extra backup register file is also used so that the processor's architectural state can be stored alternatively to one or the other of the two backup register files. This enables to keep backups of the microprocessor's state for the last two epochs. Lines in the L1 data cache will be marked (using the two volatile bits) as being either non-speculative, in the previous epoch, or in the current epoch. At the end of each epoch, the volatile bits of the previous epoch are cleared, and the tags of the current epoch are updated to indicate that

they refer to the previous epoch. During the new epoch, any access to the previous epoch's state must be first copied into the current epoch before being written, so that the previous epoch's state does not get corrupted. A similar technique for providing a sliding rollback window is described in [134].

3.1.4 System Fault Recovery

In presence of a fault, recovery to a correct microprocessor architectural state is accomplished by flushing the pipeline and copying the architectural registers from the backup register file. The memory system is protected against possible corrupted updates issued after the fault manifestation by invalidating all the cache lines marked as volatile in the local cache hierarchy. Therefore, the presence of the fault is transparent to the application's correct execution. To provide forward progress the defective module must be disabled via hardware reconfiguration.

3.1.5 Repairing the BulletProof Pipeline

In the event of a fault manifestation, the microarchitectural checkpointing mechanism will restore correct program state. However, before execution can safely continue, the underlying hardware must be repaired. The proposed technique relies on the natural hardware resource redundancy of ILP processors to reduce the cost of repair. Faulty components are removed from future operations, and the pipeline can keep running in a performance-degraded mode. To implement pipeline repair, the following facilities are included in the design:

1. Faulty functional units, such as ALUs, multipliers and decoders are disabled from further use. Consequently, further execution must limit the extent of parallelism allowed.

2. Faulty register file entries are repaired using the replacement register, as shown in Figure 3.2(b). The replacement register overrides a single entry of the register file, thus, any value read or written to the defective register is now serviced by the replacement register.

3. Faulty cache lines are excluded using a two-bit register in the LRU logic. Upon detecting a faulty line, the LRU state register is updated to indicate that the defective line is no longer eligible as a candidate line during replacement.

If the microprocessor is already impaired by many silicon defects, it may be no longer possible to tolerate an additional defect in a particular subcomponent. The degree to which

silicon defects can be tolerated is dictated by the number of redundant hardware components available. In general, with *N* components, it is possible to tolerate *N-1* defects. Once the *N-1*-th component fails, the hardware should generate a signal to the operating system to indicate that the system is no longer protected against defects. Finally, it should be noted that if the failure is the result of a transistor slowdown, *e.g.*, due to gate oxide wearout or to a negative-bias temperature instability (NBTI), it may be possible to recover the failing component by slowing down the system clock or increasing the component's voltage.

3.1.6 Handling Input/Output Requests

Instructions that perform input and output requests require special handling in the BulletProof defect tolerant microprocessor design. Since I/O operations are typically non-speculative, they can only be executed at the end of a computational epoch. To accommodate them efficiently, three flavors of I/O requests are introduced into the design: high-priority, low-priority, and speculative (the type of I/O request is associated with the memory address, and it is specified in the corresponding page table entry).

- *High priority I/O requests* are deemed extremely time sensitive, thus, they force the end of a computational epoch, which may force the processor to stall to complete the testing sweep. After this, the I/O request executes safely, and another epoch can start immediately after it.

- *Low priority I/O requests* are less time sensitive. Hence, they are held in a small queue where they age until the end of the current epoch, at which point they are all serviced. To prevent I/O starvation in programs with long computational epochs, low-priority I/O requests are only allowed to age for a small fixed period of time (about one μsec in this design). In addition, the computational epoch must end when any attempt is made to insert a low-priority request into a full I/O queue.

- *Speculative I/O requests* are I/O requests that are either insufficiently important to care about the impacts of unlikely defects (*e.g.*, writes to video RAM, which could be fixed in the next frame update), or they are idempotent (*e.g.*, the reading of a data packet from a network interface buffer). Such requests are allowed to execute speculatively before the end of a computational epoch. If a defect is encountered during the epoch in which they execute, they will just be re-executed in the following epoch, once the defective component has been disabled.

3.1.7 Assumptions and Limitations

While the BulletProof approach provides defect protection for a microprocessor pipeline and on-chip memory system at low cost with very limited performance impact, it does have a number of error model assumptions and usage limitations which are detailed below in this subsection.

In the presented BulletProof approach, a fairly treacherous error model is assumed. Specifically, it is assumed that devices can suffer from catastrophic failures at any time, which can be successfully detected with the proposed online hardware tests. In addition, transistors can suffer gradual slowdown, for example from gate oxide wearout or negative-bias temperature instability (NBTI), in which case transistors gradually slow down until they do not meet frequency requirements. In this case, the aggressive online testing clock will detect this condition before it affects computation.

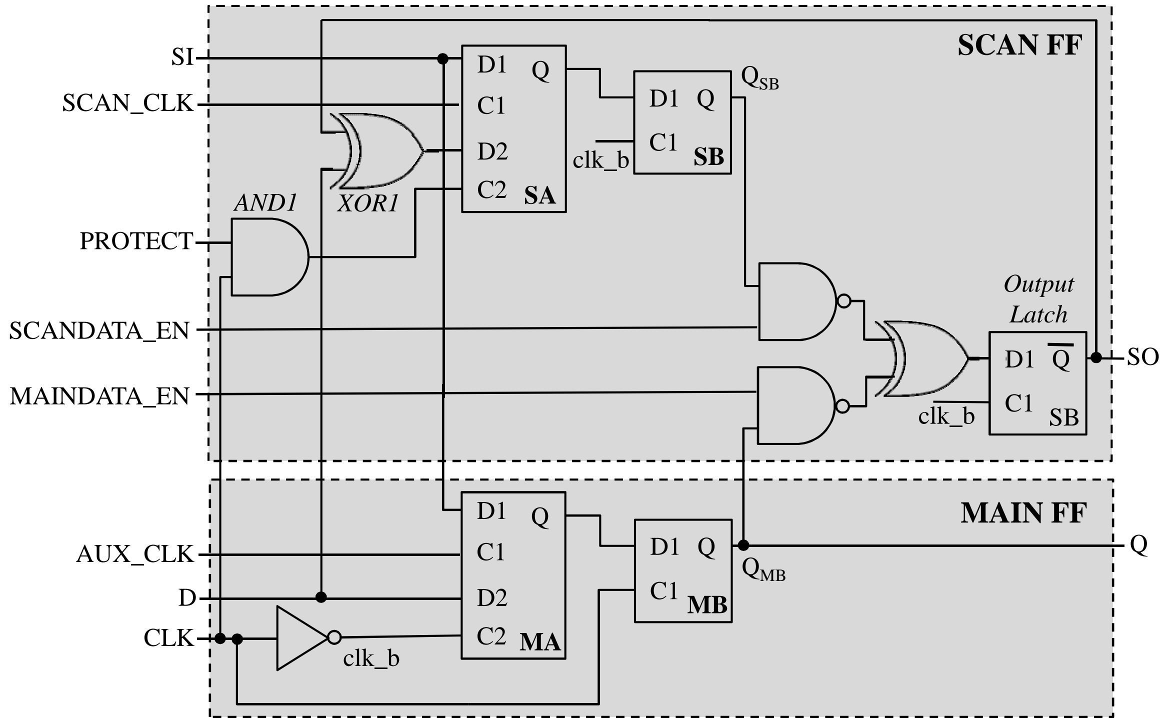
Another limitation of the BulletProof technique is that it places a few restrictions on the pipeline and on-chip cache organizations. In particular, the approach of disabling defective functional units requires multiple units of each class, otherwise, a single defect in a critical non-replicated unit could render the processor broken. Given the abundance of resources in most modern ILP processors, this limitation is not a significant drawback for most designs. Additionally, the cache organization must be set-associative to accommodate both speculative and non-speculative state.

3.2 BulletProof Protection from Transient Faults

The BulletProof techniques described so far in the previous section provide microprocessor protection only to permanent silicon defects. This section, extents the BulletProof capabilities and presents a novel circuit for transient fault detection that is based on a double-sampling scan flip-flop.

Figure 3.6 depicts the proposed fault-tolerant scan cell that is capable of detecting soft errors in both sequential and combinational logic. In addition, it can also detect permanent silicon defects in sequential elements. Figure 3.6 also lists different operating modes of the cell and their corresponding input configurations.

The BulletProof SER-tolerant flip-flop is composed of a main flip-flop block and a scan flip-flop block where each block includes a master and a slave latch. In addition, the scan flip-flop block contains an XOR gate detecting when the two master-slave flip-flops have latched different values (as it is the case when a transient fault hits) and an additional latch storing this information permanently. The two blocks are fed with two distinct clocks, the main clock and a skewed clock. In this design the skewed clock is the inverse of the main



INPUT CONFIGURATIONS FOR CELL USE

Normal op. – no protection	PROTECT	SCANDATA_EN	MAINDATA_EN	AUX_CLK
Normal op. – with protection	**0**	**0**	**0**	**0**
Shift out SI or error signal	**1**	**1**	**1**	**0**
Main FF data to scan chain	**0**	**1**	**0**	**0**
Test FF for hard failure	**0**	**1**	**1**	Single Pulse

ENERGY-DELAY CHARACTERISTICS (Normalized to a regular scan cell)

Nominal Power	**1.96**	Area	**1.5**
Power @ Error cycle	**2.69**	CLK-Q Delay	**1.12**

Figure 3.6: BulletProof SER-Tolerant Flip-Flop Design: The BulletProof SER-tolerant flip-flop design is based on double-sampling scan flip-flop. The flip-flop operates under five different operating modes depending on its input configuration.

clock and is indicated in Figure 3.6 as `clk_b`. The main flip-flop latches the incoming data signal on the positive edge of the clock, while the scan flip-flop samples the same signal on the skewed clock's positive edge. The assumption is made that transient faults manifest as glitches of less than half clock cycle duration (which is a safe assumption up to designs operating at several GHz) [88, 150]. Hence, if an incorrect value is latched in the main flip-flop due to a transient fault, the glitch will subdue before the signal is latched again half a clock cycle later by the scan block. When this situation occurs the XOR gate outputs

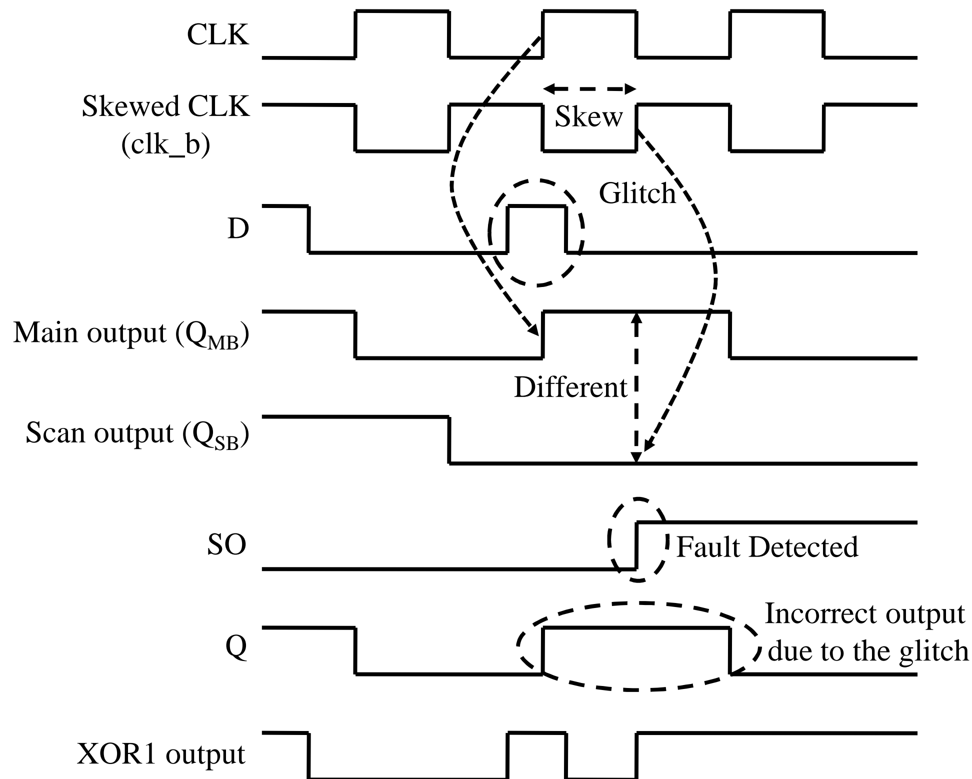


Figure 3.7: Timing Diagram of a Transient Fault Detection: The timing diagram illustrates the detection of a glitch caused by a transient fault. Once the glitch is detected, the error signal is trapped until the end of the computation epoch when all error signals are scanned out and checked.

a 1, which is stored in the `Output Latch` right away. In addition, the output signal `SO` is fed back to the `XOR1` gate, which forces the input of the scan flip-flop to always observe the complement of the data signal, continuously forcing an "SER-detected" situation.

Figure 3.7 shows a timing diagram of the situation just described. The `protect`, `scandata_en` and `maindata_en` are enabling signals which are always active during the normal protected operation. Note that in order for this flip-flop design to work, a minimum path delay constraint of 50% of the clock cycle must be enforced.

At the end of each computation epoch all error signals (SO) are shifted out through the scan chain (using the `shift out` configuration). The latches are partitioned into zones to speed up this process. If an error is detected, each cell within the zone is evaluated to discern between a transient fault or a permanent silicon latch failure. This is done using the `si`, `scan_clk` and `aux_clk` signals. If the error does not repeat, it is assumed that a transient fault had occurred, and the rollback mechanism to restore the previous known correct state is triggered. Otherwise, the cause of the error was a permanent silicon defect and hardware resource reconfiguration is triggered.

The bottom of Figure 3.6 shows the results of timing and power simulations on the error trapping cell with the skewed input clock. The output latch and the extra gates used for implementing the correct functionality account for the increase in power, area and delay in the new design, compared to a simple scan cell.

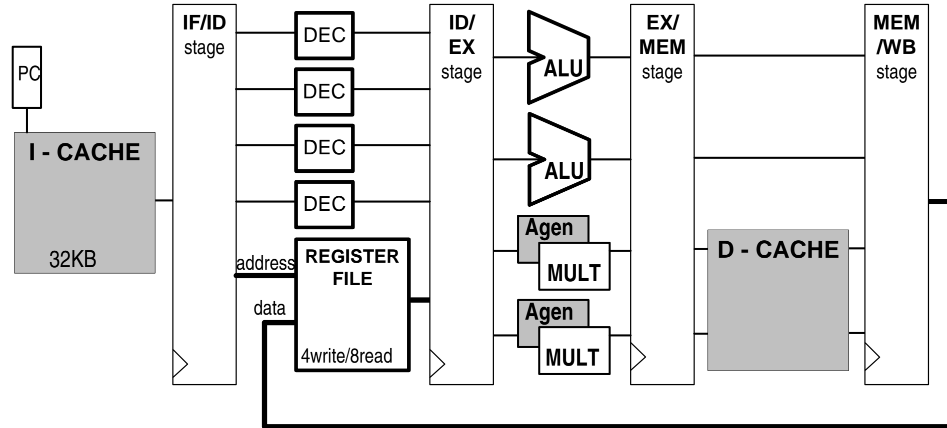
3.3 Experimental Evaluation

In this section, we evaluate the Bulletproof architecture by enhancing physical-level prototype of a 4-wide VLIW processor including instruction and data caches with the proposed BulletProof technology. The performance of the design is analyzed using both circuit timing simulation as well as architectural simulation. This enables to gauge the impacts of defect protection, both during normal operation and after a microprocessor component has been disabled. Finally, the cost of the defect protection technology is examined by measuring the area overhead of the testing logic (*e.g.*, vector generation and checkers). The defect coverage provided by the BulletProof approach, *i.e.*, what fraction of randomly placed defects are detected and successfully recovered, is also evaluated by carefully measuring the portion of the design's protected silicon area.

3.3.1 Experimental Framework

Circuit-Level Evaluation: The 4-wide VLIW prototype was specified in Verilog, and synthesized for minimum delay using the Synopsys Design Compiler. This produced a structural Verilog netlist of the processor mapped to the Artisan standard cell logic library using the TSMC 0.18um fabrication technology. The design was then placed and routed using Cadence Sedsm, which in turn yields a physical design with wire capacitances and individual component areas. Finally, the design was back annotated to obtain a more accurate delay profile, and simulated with Synposys' PrimeTime to verify its timing and functional correctness.

For each hardware component and test vector set it is verified that all stuck-at-0 and stuck-at-1 faults are detected. In general, test vector sets were identified using carefully hand-selected vectors, or by randomly cycling through random vector sets until a small group of effective vectors was located. Test vector coverage is verified by inserting a hard fault at each net of the design and then determining if a change in the output is observable for the current input test vector set. For a test vector set to provide full coverage, there must be at least one vector that identifies a hard fault in all nets of the design. Once the test vector set is identified, it is encoded into an on-chip ROM storage unit, created using Synopsys design tools.



4-wide 32-bit VLIW Processor

Five Stage Pipeline	IF/ID/EX/MEM/WB
ALU FUs/ALU Latency	2 Units/1 Cycle
LSM FUs (Load/Store/Multiply)/LSM Latency	2 Units/ 3 Cycles
L1 I-Cache/L1 D-Cache (write-back) L1 miss penalty	32KB, 4-way assoc./32KB, 4-way assoc. 10 Cycles
L2 Unified Cache (write-back)	1MB, 8-way assoc.

Figure 3.8: The BulletProof Baseline Processor: The 4-wide 32-bit VLIW microprocessor used for the evaluation of the BulletProof defect-tolerance technique.

Architectural Evaluation: The architectural evaluation was done using the Trimaran toolset, a re-targetable compiler framework for VLIW/EPIC processors [135], and the Dinero IV cache simulator [47]. The simulator was configured to model the VLIW baseline configuration and memory hierarchy as detailed in the following section. The system was evaluated using benchmarks from SPECint2000, MediaBench [69] and MiBench [43] benchmark suites. These benchmarks cover a wide range of potential applications, including desktop applications, server workloads, and embedded codes.

Coverage Analysis: Coverage analysis is measured by injecting faults into a logic timing-level simulation of the detailed VLIW processor physical design. Since characterization of silicon defects in nanometer-sized technologies is still an open research problem the stuck-at-0 and stuck-at-1 fault models were used. Defects are injected into a placed-and-routed implementation of the design. Faults are assigned to gates and wires so that the probability of a device X becoming defective p_{defect} is equal to: $p_{defect}\ \alpha\ A_x * \lambda_x$ where A_x is the area of the device and λ_x is the average estimated activity of the device. As such, large devices with high activity rates are most apt to failure, while small components or components with little activity are at lower risk.

Component	Number of Test Vectors
ALU	20
MUL	55
Decoder	63
Register File	128
Pipeline Control	12
Memory Control	13

Table 3.1: Online Test Vectors: Number of test vectors to achieve 100% coverage for stuck-at-0 and stuck-at-1 faults.

Baseline Architecture: The baseline architecture, which is enhanced with the presented BulletProof defect protection technology, is a 4-wide VLIW architecture, with a 32-KByte instruction and data caches. This architecture was chosen for the evaluation of the proposed technique because it represents a mainstream embedded target, often used in applications where cost and reliability are paramount concerns. An overview of the architecture and details of its components are shown in in Figure 3.8. The baseline pipeline is a 4-wide VLIW processor with 32-bit fixed-point datapaths. The instruction set of the processor is loosely based on the Alpha instruction set. Each VLIW instruction bundle is 128-bit long, consisting of 4 independent 32-bit instructions. The processor pipeline has five stages. The instruction fetch (IF) stage is responsible for fetching the 128-bit VLIW instruction from the 32-KByte instruction cache. The instruction decode (ID) stage decodes 4 independent instructions per cycle and fetches register operands from a register file with 8 read ports and 4 write ports. The execute (EX) stage performs arithmetic operations, multiplications, and address generation. The memory (MEM) stage accesses the 32-KByte data cache and main memory. Finally, the writeback (WB) stage retires instruction results to the register file.

3.3.2 Testing Performance and Design Coverage

In this section, the cost of the online testing infrastructure is examined. In particular, the number of vectors required to fully test each hardware component and the area cost of the hardware checkers are examined, and the overall defect coverage of the design is computed.

Online Testing Vectors: Table 3.1 lists the number of vectors needed to fully test each hardware component for stuck-at-0 and stuck-at-1 faults. The table shows that only few vectors are required to test each unit. Considering that the length of a computational epoch will typically be 1000’s of cycles, it is quite promising that testing can be completed using

Design Block	Total Area (mm^2)	Checker Area (mm^2)	% of Total Area	Protected Area (mm^2)	% of Total Area
IF	0.127	0.008	6.6	0.114	89.8
ID	0.278	0.023	8.2	0.261	96.3
RF	2.698	0.133	4.9	2.635	97.7
EX	2.993	1.166	39.0	2.896	96.8
WB	0.171	0.007	4.2	0.158	92.7
Flip-Flops	0.164	0.122	1.4	0.164	99.9
Overall Core	6.431	1.459	22.7	6.228	96.8
I-cache 32KB	2.033	0.009	0.5	1.881	92.5
D-cache 32KB	2.044	0.009	0.5	1.892	92.6
Overall System	10.508	1.477	**14.1**	10.001	**95.2**

Table 3.2: Area Overhead of the BulletProof Technique: The table reports the total area of each design block, the area dedicated to checkers, and the portion of the overall area that is protected as a result of the BulletProof technique.

only occasional idle cycles. The caches are not listed in Table 3.1 because the use of parity bits allow for the continuous detection of defects. Clearly, the time required to fully test the hardware is quite small, only 128 cycles, with the register file taking the longest time to complete its test.

Area Overhead and Design Coverage: The addition of test vector ROMs, where test vectors are stored, plus the checkers and checkpointing infrastructure bears a cost on the overall size of the design. Table 3.2 lists the total area of the defect tolerant component (Total Area), the defect protection infrastructure area (Checker Area), and the area that is covered by the test harness (Protected Area).

As shown in Table 3.2, the area overhead for defect protection is quite modest, with most overheads less than 10%. The overheads within the caches are even lower, less than 1% for the prototype. Consequently, the overall area overhead for defect protection is quite low. Adding support for defect and transient fault protection increased the total area of the design by only 14%.

The fault coverage of the BulletProof mechanism is examined by measuring the fraction of faults covered through fault injection experiments. This fraction represents the overall design defect coverage. Table 3.2 lists the coverage of the overall design, as well as the coverage of individual processor components. Overall, the design coverage is 95%, meaning that 95 out of 100 defects randomly placed into the microprocessor are covered in the BulletProof fault-tolerant design.

3.3.3 Run-time Performance

This section examines the impact of the BulletProof defect protection mechanism on the performance of programs running on the defect tolerant prototype design. The primary source of potential slowdown occurs when a computational epoch is too small (or the

Benchmark	Avg. epoch size (cycles)	Loads (%)	Stores (%)	Data L1 miss rate	Avg. ALU util. (%)	Avg. LSM util. (%)	Avg. Dec. util. (%)	Avg. reg. rw/cycle
175.vpr	50499	17.74	5.61	3.10	69.71	18.41	59.00	4.72
181.mcf	120936	21.68	3.68	3.54	36.89	10.70	67.00	5.36
197.parser	106380	22.34	13.69	2.10	54.22	19.71	52.25	4.18
256.bzip2	162508	18.78	6.39	8.88	55.91	33.93	73.50	5.88
unepic	33604	10.93	6.70	17.16	68.70	14.29	55.50	4.44
epic	196211	9.70	1.15	6.60	72.80	8.28	29.25	2.34
mpeg2dec	1135142	26.03	8.54	0.59	55.81	54.55	46.25	3.70
pegwitdec	169617	18.79	3.78	10.42	62.15	45.06	62.50	5.00
pegwitenc	304310	16.62	3.26	12.81	69.09	42.19	63.75	5.10
FFT	23145	19.18	17.89	1.49	56.88	43.95	33.50	2.68
patricia	139952	25.81	12.83	1.19	55.20	37.69	57.75	4.62
qsort	1184756	33.29	27.44	2.55	20.08	18.74	32.25	2.58
Average	**302254**	**20.07**	**9.25**	**5.87**	**56.45**	**28.96**	**52.71**	**4.22**

Table 3.3: Epoch Statistics for the Baseline Configuration: The table lists epoch statistics such as the average epoch size in cycles, along with L1 data cache miss rates, and statistics regarding the utilization of ALUs, L1 data cache memory ports (LSM), decoders, and register file ports.

testing requirements too great) to allow testing to complete within the time speculative state resources are exhausted.

Performance Impact of Defect Testing: Table 3.3 lists statistics about computational epochs for a variety of programs while running on the baseline VLIW processor with a 32 KByte 4-way set-associate data cache and an eight entry fully associative volatile victim cache. Listed is the average epoch size in cycles along with the L1 data cache miss rate. Also shown are statistics regarding the utilization of ALUs, L1 data cache memory ports (LSM), decoders, and register file ports. It is clear from this table that the performance overhead of defect testing is quite low. For the program with the shortest average epoch length (FFT), the number of test cycles is at most 0.5% of the total number of cycles within the epoch. For this program, even if the testing during idle cycles could not be completed, the performance impact would be negligible.

It should be noted that there is an interesting correlation between the epoch length and the average component utilization. For many of the programs with short epoch lengths (*e.g.*, FFT and unepic), there is also a low functional unit utilization. This is to be expected because a program with a short epoch length would have a large amount of cache turnover, which in turn would lead to many pipeline stalls and low functional unit utilization – and plenty of time for defect testing. While programs with long epochs tend to have higher component utilization, they do provide more time for the test harness to complete its task. In addition, the effect of different cache geometries on average epoch size is examined, and found that there is little performance impact for defect testing for a wide range of cache geometries.

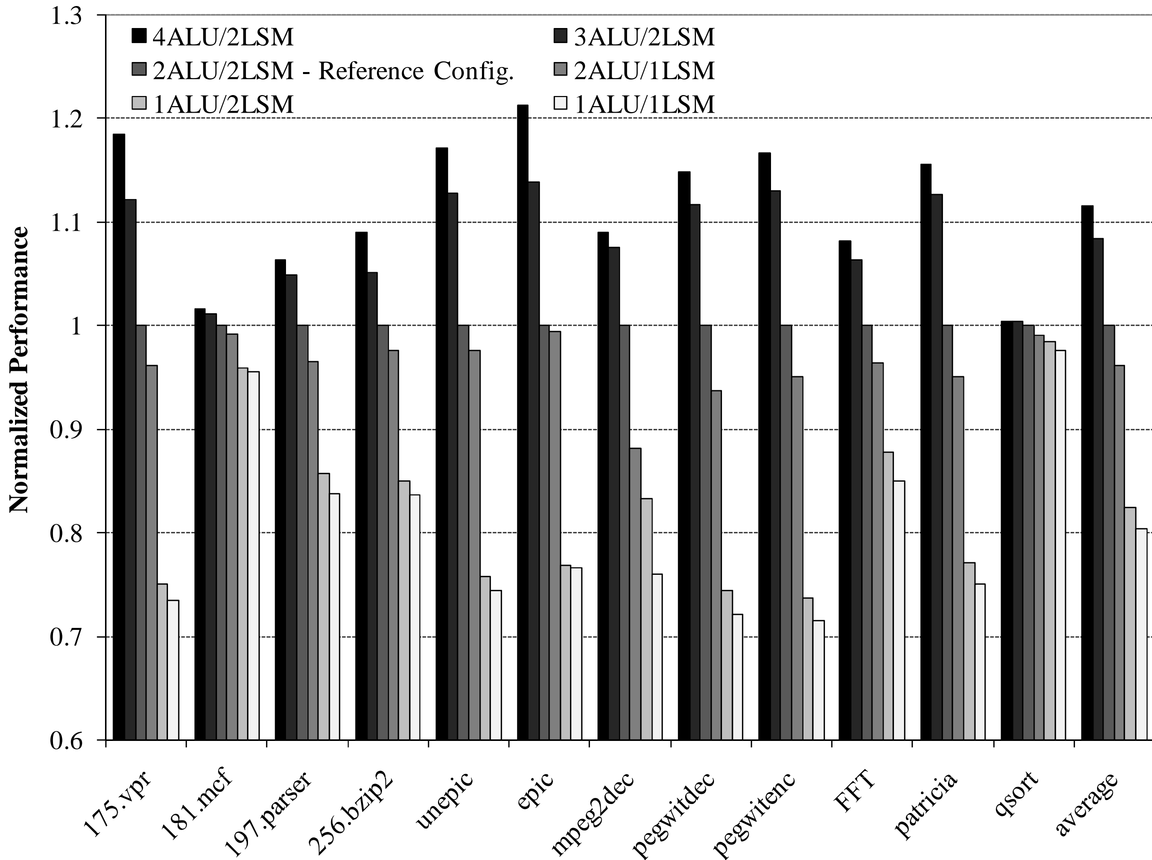


Figure 3.9: Performance Degradation of a Reconfigured BulletProof Processor: The graph shows the performance of a variety of prototype processor pipelines that have been impaired through reconfiguration. A configuration with *n-ALU/m-LSM* indicates that the prototype processor pipeline has n ALUs and m address generation/multiplier units.

Performance Impact of Degraded Mode Execution: Once a defect has been located, the microprocessor must be reconfigured by disabling the defective component. This reconfiguration will not allow as much parallelism as previously afforded in the unbroken pipeline, resulting in a performance degradation. Figure 3.9 graphs the performance of a variety of prototype processor pipelines that have been impaired through reconfiguration. In the experiments, *n-ALU/m-LSM* indicates that the experiment was run with n ALUs and m address generation units/multipliers. The number of resources is varied from one to four. As shown in Figure 3.9, losing an ALU in a 2ALU/2LSM machine configuration renders an average of 18% performance degradation. The average performance degradation is limited to only 4% when losing an address generation/multiplier unit in the same machine configuration. Machine configurations with more resources can exhibit even lower performance degradation after being impaired through resource reconfiguration. For example, machine configurations with four and three ALUs loosing one ALU results in an average performance degradation of 3% and 8% respectively.

3.4 Related Work

Defect-Tolerance Solutions: Table 3.4 compares qualitatively the BulletProof mechanism with traditional defect-tolerance solutions and more recent solutions proposed in the research literature. As discussed in Section 2.1, techniques like dual and triple modular redundancy are full-scope techniques that can provide defect coverage to the whole design with very limited intrusion to the original design. However, these hardware replication techniques lead to very high area overheads. An approach similar to hardware redundancy is N-version redundancy where the protected component is designed by *N* different groups. This approach avoids common failure modes, however it suffers from very high design cost since the replicated components are designed independently. Another traditional approach is error correction codes (ECC) that is used for detecting and correcting data corruption in memory structures and data buses. Although ECC has been proven a low overhead and effective technique to provide data protection, it is limited only to the data structures of a microprocessor design, most commonly the memory caches and the register file.

The lower part of Table 3.4, compares the BulletProof approach [116, 83] with more recent mechanisms found in the research literature, listed in chronological order with the less recent at the top. The first work that proposed a comprehensive approach for microprocessor tolerance to silicon defects was DIVA, proposed by Austin in [6]. DIVA is a simple online checker component inserted into the retirement stage of a complex out-of-order microprocessor pipeline that continuously validates the computation, communication, and control exercised in the microprocessor core [6, 143]. 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. The hardware overhead of a DIVA checker is estimated to be around 6% of a full complex out-of-order microprocessor, which compared to the traditional hardware replication techniques is extremely low. However, augmenting a complex microprocessor design with a DIVA checker has a higher design complexity and it is more intrusive than the traditional hardware replication techniques. Furthermore, a limitation of the DIVA approach is that it does not diagnose the root cause of an error in order to repair the underlying hardware and prevent the error from occurring again.

Next, Bower *et al.* [19] proposed a hardware mechanism for self-repairing array structures to provide defect detection and repair capabilities for microprocessor array structures such as the reorder buffer and branch history table. The proposed mechanism detects silicon defects by employing dedicated “check rows”. Every time an entry is written to the

Defect Tolerance Solution	Defect Coverage	Area Overhead	Performance Overhead	Design Intrusion/ Complexity	Comments
Traditional Solutions					
Dual Modular Redundancy (DMR)	Very High (~99%)	Very High (>100%)	Very Low (<5%)	Low	Provides only error detection. Easy to cover the whole design.
Triple Modular Redundancy (TMR)	Very High (~99%)	Ultra High (>200%)	Very Low (<5%)	Low	Provides both error detection and forward recovery. Easy to cover the whole design.
N-Version Redundancy	Very High (~99%)	Very High (>100%)	Very Low (<5%)	Very High	*N* different versions of the component have to be implemented.
Error Correction Codes (ECC)	Memory Structures	Medium (~15%)	Very Low (<5%)	Low	Limited only to memory structures or data buses.
Research-Stage Solutions					
DIVA Austin [6]	Not Available	Low (~6%)	Not Available	Medium	Uses an online checker at the pipeline's retirement stage.
SRAS Bower *et al.* [19]	Only Array Structures	Not Available	Not Available	Medium	Limited to array structures. Requires hardware changes in the array structures.
Bower *et al.* [20]	Not Available	Medium (>15%)	Not Available	High	Uses DIVA checkers and pipeline additions that truck instruction execution for defect diagnosis.
BulletProof [116, 83]	High (~95%)	Medium (~14%)	Ultra Low (<1%)	Medium	Uses BIST-like on-chip hardware checkers.
ElastIC Sylvester *et al.* [129]	Under Development/Evaluation			High	Uses on-chip sensors, silicon wear-out prediction units, and on-chip testers.
Argus Meixner *et al.* [85]	High (~98%)	Medium (~11%)	Low (~4%)	Medium	Uses runtime checkers for the validation of control flow, computation, dataflow, and memory operations.
StageNet Gupta *et al.* [42]	Not Available	Medium (~15%)	Medium (~10%)	High	Pipeline stages need to be isolated and connected through crossbar switches. No error detection support.

Table 3.4: Comparing BulletProof To Related Work: Comparison of BulletProof to traditional defect-tolerance solutions and more recent techniques found in the research literature. The techniques are compared in respect to their defect coverage, area overhead, runtime performance overhead, and the degree they intrude in the original design and they are presented in chronological order with the less recent at the top.

array structure, the same data is also written into a check row. Then, both locations are read out and their values are compared to detect defective rows. To repair defective arrays, the mechanism exploits the inherent resource redundancy of these structures and redirects any accesses to defective rows to other functionally correct rows. Although the area overhead of the technique is expected to be low, its implementation requires hardware design changes to the protected array structures. Furthermore, the technique is limited only to array structures and it does not cover the other resources of the microprocessor.

To address this limitation, Bower *et al.* extended their work in [20], where they proposed a fault-tolerant microprocessor design that uses DIVA checkers for system-level error detection coupled with a mechanism for diagnosing silicon defects by tracking the instruction occupancy through the microprocessor's pipeline. This mechanism covers the

whole microprocessor pipeline and after diagnosing a silicon defect in one of the pipeline components, the microprocessor reconfigures (*i.e.*, disables) the defective part and continues operation at a gracefully degraded level of performance. The estimated area overhead of this solution is around 15% of the pipeline area.

In [129], Sylvester *et al.* proposed the ElastIC architecture which uses in-situ sensors in combination with reliability and power models to predict the lifetime and wearout of the underlying hardware. This approach enables the dynamic trade-off of performance with longer lifetime and reliability using dynamic voltage scaling techniques. The prototype and evaluation process for this approach is currently in progress and there are not yet known estimates for its coverage, area, or performance overhead.

More recently, Meixner *et al.*, in [85], presented Argus, an error detection technique for simple processor cores. The Argus technique continuously checks invariants to detect execution errors, without the need for redundant computation. Specifically, Argus, uses run-time invariant checking in four fundamental tasks: the control flow, the dataflow, computation, and memory access. An implementation of the Argus system, the Argus-1, that illustrates the engineering trade-offs between checker costs and error coverage was presented in [85]. The Argus-1 prototype implementation was based on a single-issue, 4-stage, in-order processor and is characterized by a 11% area overhead and around 4% runtime performance overhead. The Argus approach, like the BulletProof approach, provides error detection for errors caused by both permanent silicon defects and transient faults and offers an alternative low-cost defect-tolerance approach compared to the traditional defect-tolerance approaches. However, Argus incurs a slightly higher runtime performance overhead than the BulletProof approach.

Finally, in [42], Gupta *et al.* presented StageNet, a highly reconfigurable multicore architecture. StageNet is a reconfigurable multicore computing substrate designed as a network of pipeline stages, rather than isolated cores in a chip-multicore processor. The StageNet network is formed by replacing the direct connections at each pipeline stage boundary by a crossbar switch. Within the StageNet network, pipeline stages can be selected dynamically from the pool of available stages to form logical processing cores, thus permanent silicon failures can be easily isolated by adaptively routing around defective stages. The StageNet and the BulletProof approaches can be considered complementary techniques, since the BulletProof framework can efficiently detect and diagnose silicon defects in the microprocessor, and the StageNet substrate can effectively reconfigure the microprocessor's hardware resources and repair the microprocessor design.

From Table 3.4, we observe that the BulletProof mechanism is characterized by extremely low runtime performance overhead, but the provided defect coverage is lower

than some of the most recently proposed techniques and has higher area overhead. Based on this observation, we considered to trade-off runtime performance overhead with higher defect coverage and lower area cost. To make this trade-off possible, we found that we could move the defect detection and diagnosis process form the BulletProof's on-chip hardware checkers to software testing routines. The movement of the hardware checking process from hardware to software is the topic of the next chapter, Chapter IV.

Transient Fault Tolerance Solutions: Several approaches for providing transient fault tolerance have been proposed in the past few years. The concept of using time redundancy methods for mitigating soft errors has been explored in [5], [97] and [81]. In particular, in [81], three samples of the input are taken at different clock edges and the final output is determined using a majority voter.

An approach closer to the technique used by BulletProof to tolerate transient faults is presented in [90]. In [90], Mitra *et al.* propose reusing scan chain resources for transient fault detection in flip-flops. They introduce two different scan cell designs which are based on blocking and trapping transient faults at the output of each flip-flop. While their approach is efficient in terms of area, power and delay overhead, it does not detect transient faults in combinational logic. The solution in [91] proposes a time-redundancy based scheme with scan-path reuse in which a time-shifted version of the input is given to the scan flip-flop. The C-element which was introduced in [90] is then used to block the error at the flip-flop's output.

3.5 Chapter Summary

This chapter presented BulletProof, a low cost technique that protects a microprocessor pipeline and caches against transient faults and permanent silicon defects. The approach taken by BulletProof is notably different from traditional approaches to fault tolerance. A microarchitectural checkpointing mechanism creates speculative epochs of computation after which distributed, domain-specific on-line checkers run BIST-like tests to verify the integrity of the underlying hardware. Additionally, a double-sampling latch design is used to detect transient fault logic glitches which have corrupted the pipeline state. If, at the end of an epoch, the hardware is fault-free, the epoch computation is allowed to retire to non-speculative state. In the event that a fault is exposed, the program state is rolled back to the last known good program state at the beginning of the last epoch. If the fault is due to a transient fault, the epoch is re-executed, otherwise, the defective component is disabled, thereby allowing the processor to continue correct execution in a degraded-performance mode.

A 4-wide VLIW physical-level prototype processor enhanced with the BulletProof low-cost solution for fault tolerance was implemented. Analysis of this design indicates that area overhead of the BulletProof mechanism is quite modest, providing transient and hard silicon fault protection with only a 14% increase in total area. This is a remarkable improvement over traditional redundancy-based techniques, such as triple-modular redundancy, which incurs overheads starting at 200%. Additionally, it was demonstrated through gate-level fault injection studies that fault-detection coverage is very high: 95% of all hard silicon defects and 99% of all transient faults are covered. Additional simulation studies confirmed that periodic online testing has negligible impact on the overall system performance. Additionally, we examine the performance of prototype processors running with disabled components in a degraded mode. When a 4-wide VLIW lost only one resource, performance impacts were limited to only a 6% slowdown. Larger impacts were seen by the loss of a single resource in 2-wide VLIW processor, resulting in an overall slowdown of 26%.

The BulletProof technique makes a strong case for the use of online periodic hardware checking coupled with microarchitectural checkpointing to implement future defect tolerant microprocessors. The approach is both efficient, with high coverage and low performance impacts, and also inexpensive, with small area overhead.

CHAPTER IV

A Software-Based Periodic Hardware Checking Solution - The ACE Framework

The BulletProof approach, presented in the previous chapter, demonstrated that periodic hardware checking techniques can provide the same reliability guarantees as traditional defect-tolerance solutions that continuously monitor the execution for errors through redundant computation, but at a much lower cost. However, even in the BulletProof approach, there is a need for some additional hardware resources needed to perform the periodic hardware checking. This need, leads to some additional hardware overhead which in the case of the BulletProof prototype examined in the previous chapter, it was observed to reach a 14% area overhead over the whole processor design. Furthermore, in BulletProof, in order to lower the silicon cost, the testers were customized to the tested modules, a design decision that lead to increased design complexity as a specialized tester needed to be designed for each module covered by BulletProof. In addition, the majority of the BulletProof online hardware checkers used BIST-like testing techniques that bind a specific testing approach (*e.g.*, fault model) into silicon and cannot be modified or adapted in the field while the processor is operating.

To address the limitations observed in the BulletProof mechanism, this chapter introduces the *Access-Control Extension (ACE) Framework*, a software-based technique that shifts the silicon defect detection and diagnosis process from on-chip hardware checkers into software. In the ACE framework, the hardware provides the necessary substrate to facilitate the hardware testing, and the software makes use of this substrate to perform the hardware testing. The ACE framework addresses the limitations of the BulletProof approach by: 1) it effectively removes the need for on-chip hardware checkers and moves this functionality to software, 2) it is not hardwired in the design and therefore has ample flexibility to be modified/upgraded in the field, 3) it can be uniformly applied to any microprocessor module with low design complexity because it does not require module-specific customizations, and 4) it provides wider coverage across the whole chip, including non-core modules.

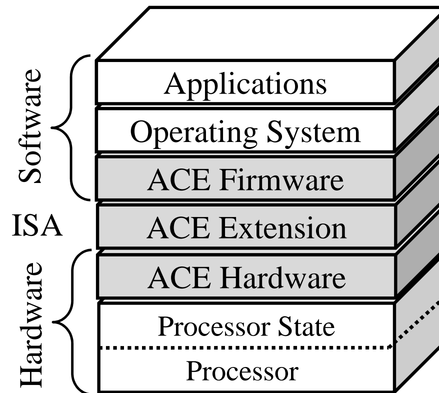


Figure 4.1: ACE Framework Overview: The ACE framework fits in the hardware/software stack below the operating system.

The ACE framework introduces specialized Access-Control Extension (ACE) instructions that are capable of accessing and controlling virtually any portion of the microprocessor's internal state. Special firmware periodically suspends microprocessor execution and uses the ACE instructions to run directed tests on the hardware and detect if any component has become defective. To provide faster and more flexible software access to different microarchitectural components at low hardware overhead, the ACE framework leverages the pre-existing scan-chain infrastructure [67] that is conventionally integrated in existing microprocessor designs and used during manufacturing testing.

Figure 4.1 shows how the ACE framework fits in the hardware/software stack below the operating system layer. The ACE framework provides particularly wide coverage, as it not only tests the internal processor control and instruction sequencing mechanisms through software functional testing, but it can also check all datapaths, routers, interconnect and microarchitectural components by issuing ACE instruction test sequences. Additionally, the ACE framework provides a complete defect-tolerance solution by incorporating its defect detection and diagnosis capabilities in a coarse-grained checkpointing and recovery environment.

In the remainder of this chapter, Section 4.1 introduces a novel set of instructions, called Access-Control Extension (ACE), that can access and control the microprocessor's internal state. This set of instructions can be used by special firmware that periodically suspend microprocessor execution to run directed tests on the hardware. Section 4.2 provides the methodology used to experimentally evaluate the ACE framework, and Section 4.3 presents an analysis of the results of the ACE framework's experimental evaluation on a commercial chip-multiprocessor based on Sun's Niagara. Section 4.4 provides previous research work that is related with the ACE framework, and the chapter is summarized in Section VII.

4.1 Software-Based Periodic Defect Detection and Diagnosis

A key challenge in implementing a software-based defect detection and diagnosis technique is the development of effective software routines to check the underlying hardware. Commonly, software routines for this task suffer from the inherent inability of the software layer to observe and control the underlying hardware, resulting in either excessively long test sequences or poor defect coverage. Current microprocessor designs allow only minimal access to their internal state by the software layer; often all that software can access consists of the register file and a few control registers (such as the program counter (PC), status registers, *etc.*). Although this separation provides protection from malicious software, it also largely limits the degree to which stock hardware can utilize software to test for silicon defects.

An example scenario where the lack of observability compromises the efficiency of software testing routines is a defective reorder buffer entry. In this scenario, a software-based solution can detect such a situation only when the defect causes an error that propagates to an accessible state, such as the register file, memory, or a primary output. Moreover, without specific knowledge as to how the architectural state was corrupted, the *diagnosis* of the source cause of the erroneous result is very challenging.[1]

To overcome this limited accessibility, the software-based framework presented in this chapter employs architectural support through an extension to the processor's ISA. Specifically, the ISA extension adds a set of special instructions that enable full observability and control of the hardware's internal state. These Access-Control Extension (ACE) instructions are capable of reading/writing from/to any part of the microprocessor's internal state. ACE instructions make it possible to probe underlying hardware and systematically and efficiently assess if any hardware component is defective.

4.1.1 An ACE-Enhanced Architecture

A microprocessor's state can be partitioned into two parts: accessible from the software layer (*e.g.*, register file, PC, *etc.*), or not (*e.g.*, reorder buffer, load/store queues, *etc.*). An ACE-enhanced microarchitecture allows the software layer to access and control (almost) all of the microprocessor's state. This is done by using *ACE instructions* that copy a value from an architectural register to any other part of the microprocessor's state, and *vice versa*.

This approach inherently requires the architecture to access the underlying microarchitectural state. To provide this accessibility without a large hardware overhead, we leverage

[1]The sole fact that a hardware fault had propagated to an observable output does not provide information on where the defect originated.

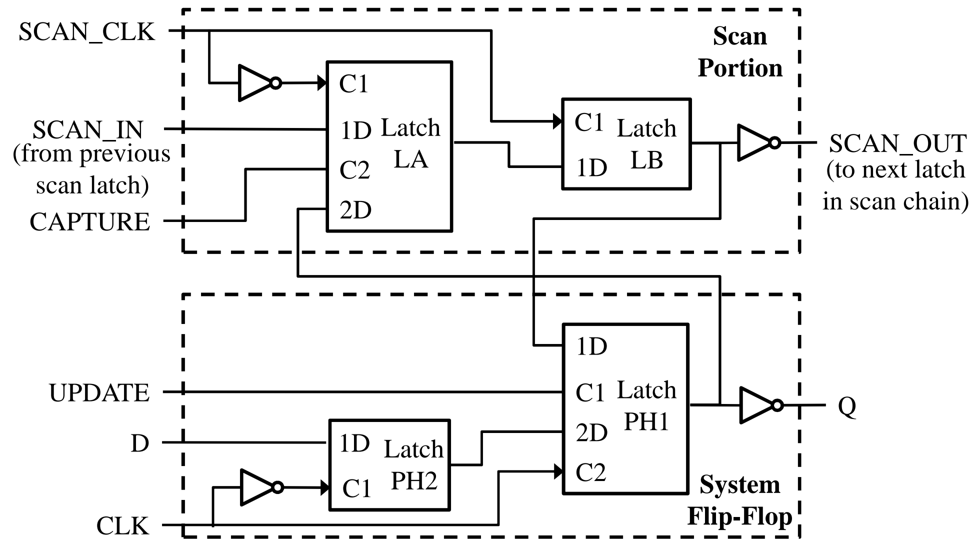


Figure 4.2: A Typical Scan Flip-Flop: The presented scan flip-flop is adapted from [90]. The system flip-flop is used during the normal operation of the microprocessor, while the scan portion is used during testing to shift in and out test patterns and test responses. The ACE framework leverages the microprocessor's scan architecture to facilitate online testing.

the existing scan chain infrastructure. Most modern processor designs employ full hold-scan techniques to aid and automate the manufacturing testing process [67, 146]. In a full scan design, each flip-flop of the processor state is substituted with a scan flip-flop and connected to form one or more shift registers (scan chains) [23]. Figure 4.2 shows a typical scan flip-flop design [90, 67]. The system flip-flop is used during the normal operating mode, while the scan portion is used during testing to load the system with test patterns and to read out the test responses. The ACE framework extends the existing scan-chain using a hierarchical, tree-structured organization to provide fast software access to different microarchitectural components. The scan chain is operated at-speed, *i.e.,* at the same frequency as the processor clock, to facilitate online testing, as in some modern microprocessors [75].

ACE Domains and Segments: In the ACE extension implementation, the microprocessor design is logically partitioned into several *ACE domains*. An ACE domain consists of the state elements and combinational logic associated with a specific part of the microprocessor. Each ACE domain is further subdivided into *ACE segments*, as shown in Figure 4.3(a). Each ACE segment includes only a fixed number of storage bits, which is the same as the width of an architectural register.

ACE Instructions: Using this hierarchical structure, ACE instructions can read or write any part of the microprocessor's state. Table 4.1 shows a description of the ACE instruction set extensions.

ACE_set $src,<ACE Domain#>,<ACE Segment#> Copy *src* register to the scan state (scan portion)
ACE_get $dst,<ACE Domain#>,<ACE Segment#> Load scan state to register *dst*
ACE_swap <ACE Domain#>,<ACE Segment#> Swap scan state with processor state (system FFs)
ACE_test: Three cycle atomic operation. Cycle 1: Load test pattern, Cycle 2: Execute for one cycle, Cycle 3: Capture test response
ACE_test <ACE Domain#>: Same as ACE_test but local to the specified ACE domain

Table 4.1: The ACE Instruction Set Extensions: The ACE instructions can copy a value from an architectural register to any other part of the microprocessor's state, and *vice versa*.

ACE_set copies a value from an architectural register to the scan state (scan portion in Figure 4.2) of the specified ACE segment at-speed (*i.e.*, at the processor's clock frequency). Similarly, ACE_get loads a value from the scan state of the specified ACE segment to an architectural register at-speed. These two instructions can be used for manipulating the scan state through software-accessible architectural state. The ACE_swap instruction is used for swapping the scan state with the processor state (system flip-flops) of the ACE segment by asserting both the UPDATE and the CAPTURE signals (see Figure 4.2).

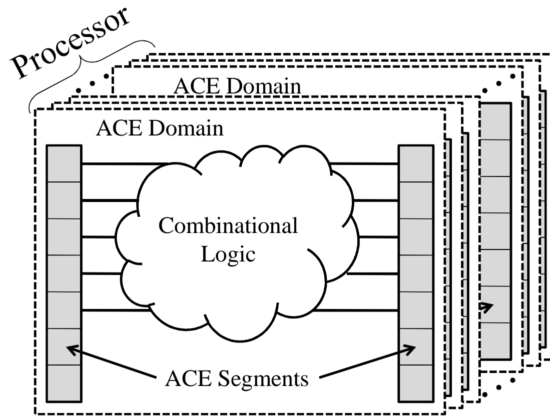
Finally, ACE_test is a test-specific instruction that performs a three-cycle atomic operation for orchestrating the actual testing of the underlying hardware (see Section 4.1.2 for an example). ACE_test is used after the scan state is loaded with a test vector using the ACE_set instruction. In the first cycle, the scan state is swapped with the processor state. The second cycle is the actual test cycle in which the processor executes for one clock cycle.[2] In the third cycle, the processor state is swapped again with the scan state. The last swap restores the processor state in order to continue normal execution and moves the test response back to the scan state where it can be validated using the ACE_get instruction. Another version of ACE_test takes as argument an ACE domain index, which allows testing to be performed locally only in the specified domain.[3]

ACE Tree: During the execution of an ACE instruction, data needs to be transferred from the register file[4] to any part of the chip that contains microarchitectural state. In order

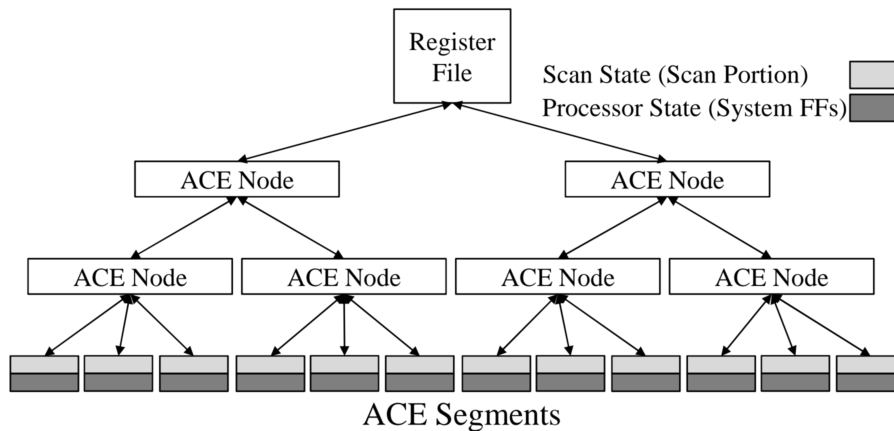
[2]Note that this is analogous to single-stepping in software debugging.

[3]ACE_test is logically the same as an atomic combination of ACE_swap, followed by a single test cycle, followed by another ACE_swap.

[4]Either from general-purpose architectural registers or from special-purpose architectural registers.



(a)



(b)

Figure 4.3: The ACE Architecture: In part (a), the chip is logically partitioned into multiple ACE domains. Each ACE domain includes several ACE segments. The union of all ACE segments comprises the full chip's state (excluding SRAM structures). In part (b), data is transferred from/to the register file to/from an ACE segment through the bidirectional ACE tree.

to avoid long interconnect, which would require extra repeaters and buffering circuitry, the data transfer between the register file and the ACE segments is pipelined through the *ACE tree*, as shown in Figure 4.3(b). At the root of the ACE tree is the register file while the ACE segments are its leaves. At each intermediate tree level there is an *ACE node* that is responsible for buffering and routing the data based on the executed operation. The ACE tree is a bidirectional tree allowing data transfers from the register file to the ACE segments and back. By designing the ACE tree as a balanced tree (all paths have the same length), each ACE instruction that reads/writes any segment of the microprocessor state takes the same number of clock cycles (*i.e.*, the tree's depth). Note that ACE instructions can be executed in a pipelined fashion over the ACE tree.

In a uniprocessor system the ACE topology is the simplest possible, since it consists of a single ACE tree rooted at the processor's register file. However, CMP systems consisting of several cores, on-chip caches, and supporting modules such as memory controllers and cross-bars (*i.e.*, *non-core* modules), might require more complex ACE topologies. In CMP systems it is possible to design multiple ACE trees, each originating from a distinct register file of the multiple cores in the system. Since non-core modules usually do not have instruction execution capabilities, they cannot include an ACE tree of their own. Therefore, in the ACE framework implementation, each core's ACE tree spans over the core's resources as well as over non-core modules.

In order to avoid any malicious use of the ACE infrastructure, ACE instructions are privileged instructions that can be used only by *ACE firmware*. ACE firmware routines are special applications running between the operating system layer and the hardware in a trusted mode, similarly to other firmware, such as device drivers. Each microprocessor vendor can keep the specific mapping between the microprocessor's state and the ACE domains/segments as classified information for security reasons. Therefore, it is expected that ACE firmware will be developed by microprocessor vendors and distributed to the customers.

Design Complexity: Since the ACE Tree is a regular structure that routes data from the register file to the scan chains and vice versa, its implementation and insertion into the microprocessor implementation can be automated by CAD tools, similar to the way that scan chains are automatically implemented and inserted in current microprocessors today. The main intrusive portion of the ACE Tree that needs interaction with existing processor components are the additional read/write ports needed to connect the root of the ACE Tree to the processor register file. Similarly, the ACE instruction set extensions are likely not intrusive to the microarchitecture since their operations are relatively simple and their implementation does not affect the implementation of other instructions in the ISA.

4.1.2 ACE-Based Online Testing

ACE instruction set extensions make it possible to craft programs that can efficiently and accurately detect underlying hardware defects. The approach taken in building test programs, however, must have high-coverage, even in the presence of defects that might affect the correctness of ACE instruction execution and test programs. This section describes how test programs are designed.

ACE Testing and Diagnosis: Special firmware periodically suspends normal processor execution and uses the ACE infrastructure to perform high-quality testing of the underlying hardware. A test program exercises the underlying hardware with previously

```
Step 1: Test Pattern Loading

// load test pattern to scan state
for(i=0;i<#_of_ACE_Domains;i++){
  for(j=0;j<#_of_ACE_Segments;j++){
    load    $r1,pattern_mem_loc
    ACE_set $r1, i, j
    pattern_mem_loc++
}
```

```
Step 2: Testing

// Three cycle operation
// 1)load test pattern
// to processor state
// 2)execute for one cycle
// 3)capture test response &
// restore processor state
ACE_test
```

```
Step 3: Test Response Validation

// validate test response
for(i=0;i<#_of_ACE_Domains;i++){
  for(j=0;j<#_of_ACE_Segments;j++){
    load    $r1,test_resp_mem_loc
    ACE_get $r2, i, j
    if ($r1!=$r2) then ERROR else
    test_resp_mem_loc++
}
```

Figure 4.4: ACE Firmware: Pseudo-code for 1) loading a test pattern, 2) testing, and 3) validating the test response.

generated test patterns and validates the test responses. Both the test patterns and the associated test responses are stored in physical memory. The pseudo-code of a firmware code segment that applies a test pattern and validates the test response is shown in Figure 4.4. First, the test program stops normal execution and uses the `ACE_set` instruction to load the scan state with a test pattern (Step 1). Once the test pattern is loaded into the scan state, a three-cycle atomic `ACE_test` instruction is executed (Step 2). In the first cycle, the processor state is loaded with the test pattern by swapping the processor state with the scan state (as described in the previous section). The next cycle is the actual test cycle where the combinational logic generates the test response. In the third cycle, by swapping again the processor state with the scan state, the processor state is restored while the test response is copied to the scan state for further validation. The final phase (Step 3) of the test routine uses the `ACE_get` instruction to read and validate the test response from the scan state. If a test pattern fails to produce the correct response at the end of Step 3, the test

program indicates which part of the hardware is defective[5] and disables it through system reconfiguration [114, 29]. If necessary, the test program can run additional test patterns to narrow down the defective part to a finer granularity.

Given this software-based testing approach, the firmware designer can easily change the level of defect coverage by varying the number of test patterns. As a test program executes more patterns, coverage increases. Automatic test pattern generation (ATPG) tools [23] can be used to generate compact test pattern sets adhering to specific fault models.

Basic Core Functional Testing: When performing ACE testing, there is one initial challenge to overcome: ACE testing firmware relies on the correctness of a set of basic core functionalities which load test patterns, execute ACE instructions, and validate the test response. If the core has a defect that prevents the correct execution of the ACE firmware, then ACE testing cannot be performed reliably. To bypass this problem, specific programs to test the basic functionalities of a core before running any ACE testing firmware are employed. If these programs do not report success in a timely manner to an independent auditor (*e.g.*, the operating system running on the other cores), then we assume that an irrecoverable defect has occurred on the core and it is permanently disabled. If the basic core functionalities are found to be intact, finer-grained ACE testing can begin. Although these basic functionality tests do not provide high-quality testing coverage, they provide enough coverage to determine if the core can execute the targeted ACE testing firmware with a very high probability. A similar technique employing software-based functional testing was used for the manufacturing testing of Pentium 4 [99].

Testing Frequency: Device experts suggest that the majority of wearout-related defects manifest themselves as progressively slow devices before eventually leading to a permanent breakdown [15, 71]. Therefore, the initial observable symptoms of most wearout-related defects are timing violations. To detect such wearout-related defects early, we employ a test clock frequency that is slightly faster than the operating frequency. Specifically, the existing dynamic voltage/frequency scaling mechanisms employed in modern processors [79] can be extended to support a frequency that is slightly higher than the fastest used during normal operation.[6]

[5]By interpreting the correspondence between erroneous response bits and ACE domains.

[6]The safeguard margins used in modern microprocessors (to tolerate process variation) allow the use of a slightly faster testing frequency with a negligible number of false positives [33].

4.1.3 ACE Testing in a Checkpointing and Recovery Environment

The ACE testing framework is incorporated within a multiprocessor checkpointing and recovery mechanism (*e.g.*, SafetyNet [120] or ReVive [101]) to provide support for system-level recovery. When a defect is detected, the system state is recovered to the last checkpoint (*i.e.*, correct state) after the system is repaired.

In a checkpoint/recovery system, the release of a checkpoint is an irreversible action. Therefore, the system must execute the ACE testing firmware at the end of each checkpoint interval to test the integrity of the whole chip. A checkpoint is released only if ACE testing finds no defects. With this policy, the performance overhead induced by running the ACE testing firmware depends directly on the length of the checkpoint interval, that is, longer intervals lead to lower performance overhead. The trade-off between checkpoint interval size and ACE testing performance overhead is explored in Section 4.3.5.

To achieve long checkpoint intervals, I/O operations need to be handled carefully. I/O operations such as filesystem/monitor writes or network packet transmissions are irreversible actions and can force an early checkpoint termination. Premature checkpoint terminations can be avoided by buffering I/O operations as described in [95]. Alternatively, the operating system can be modified to allow speculative I/O operations as described in [98]. Section 4.3.7 evaluates the effect of frequent I/O operations on the performance overhead of our technique.

4.1.4 Putting it Together: Algorithmic Flow of ACE-Based Testing

Table 4.2 shows the flow of ACE-Based Online testing in a checkpointing and recovery environment with single-threaded execution. Other execution models are examined in the next section. Two points are worth noting in the algorithm. First, a lightweight context switch is performed from the application thread to the ACE testing thread at the beginning of the test and vice versa at the end of the test. Lightweight context switching [2, 65] in a single cycle is supported by many simultaneously-multithreaded processors today, including Sun's UltraSPARC T1. If lightweight context switch support is not available, then a pipeline flush is required. Our results show that context switch penalty, even if it is hundreds of cycles, only negligibly increases the overhead of ACE testing. Second, if either the basic core functional test or the ACE firmware test fails, ACE testing firmware disables the tested core and traps to system software.[7] If the ACE firmware test fails, the

[7] Note that if a certain test takes longer than an unreasonably long time interval (*i.e.*, greater than 10 times the maximum latency of the performed test), a watchdog timer detects this and repeats the test. If the test fails or times out twice, then an irrecoverable core defect is assumed and the microprocessor traps to system software.

Step	Action
1	Run regular application thread until the checkpointing interval is reached
2	Lightweight context switch to ACE testing mode
3	Run basic core functional test (described in Sections 4.1.2 and 4.3.1)
3-Fail	If the functional test fails twice, declare fault, disable core, and trap to system software for analysis and recovery
4	Run the ACE firmware (shown in Figure 4.4)
4-Fail	If ACE firmware results in ERROR, declare fault, disable core, and trap to system software for analysis and recovery
5	Discard old checkpoint, create new checkpoint, context switch back to regular application thread; go to Step 1

Table 4.2: Algorithmic Flow of ACE-Based Testing: During ACE-based testing, a lightweight context switch is performed from the application thread to the ACE testing thread at the beginning of the test and vice versa at the end of the test. If either the basic core functional test or the ACE firmware test fails, ACE testing firmware disables the tested core and traps to system software.

system software performs defect diagnosis to localize the defect. To do so, the system software maps the ACE segments that fail to match the expected test response to specific hardware components (*i.e.*, the combinational logic driving the flip-flops of the ACE segments). If reconfigurability support is provided within those hardware components, the ACE firmware can pinpoint these components to be disabled.

4.1.5 ACE Testing Execution Models

Single-Threaded Sequential ACE Testing: The simplest execution model for ACE testing is to invoke the ACE testing process at the end of each checkpoint interval. In this execution model, the application runs normally on the processor until the buffering resources dedicated to the checkpoint are full and a new checkpoint needs to be taken. At this point, a context switch between the application process and the ACE testing process happens. If the ACE testing routine deems the underlying hardware defect-free, a new checkpoint of the processor state is taken and the execution of the application process is resumed. Otherwise, system repair and recovery are triggered. Figure 4.5(a) illustrates this single-threaded sequential execution model.

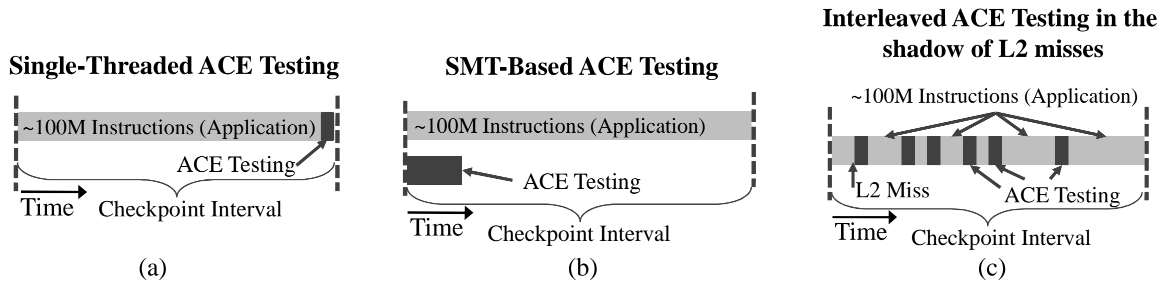


Figure 4.5: Different ACE Testing Execution Models: Part (a) illustrates ACE testing in a single-threaded sequential execution model where the ACE testing thread is executed exclusively after application execution. In part (b) the ACE testing thread runs simultaneously with the application in a 2-way SMT execution environment. In part (c) ACE testing is interleaved with application execution and runs in the shadow of L2 cache misses.

SMT-Based ACE Testing: In processors that support simultaneous multithreading (SMT) execution [112, 48, 136], it is possible for the ACE firmware to run simultaneously with the application threads running on separate execution contexts. This execution model is illustrated in Figure 4.5(b) and could be higher performance since it overlaps the latency of ACE testing with actual application execution.

Fortunately, the majority of the instructions used by the ACE testing firmware do not entail any synchronization requirements between the ACE testing thread and the other threads running on the processor. For example, the ACE instructions used to load a test pattern into the scan state (`ACE_set`) or read and validate a test response (`ACE_get`) do not affect the execution of other threads running on the processor. The work performed by these instructions can be fully overlapped with application execution.

However, the `ACE_test` instruction momentarily changes the microarchitectural state of the entire processor and thus affects the normal execution of all running threads. To avoid the incorrect execution of other running threads, when an `ACE_test` instruction is executed by the ACE testing thread, all other threads need to pause execution. This is implemented by using simple synchronization hardware that pauses execution of all other threads (*i.e.*, stalls their pipelines) when an `ACE_test` instruction starts execution and resumes their execution once the test instruction is completed. Notice that during testing, the processor's microarchitectural state is stored in the scan state. The microarchitectural state gets restored right after the test cycle (see Section 4.1.1) enabling the seamless resumption of normal processor execution.

The advantage of the SMT-based ACE testing model is its lower performance overhead compared to single-threaded sequential ACE testing. The disadvantage is that this model requires a separate SMT context to be present in the underlying processor.

Interleaved ACE Testing in the Shadow of L2 Misses: When the ACE testing thread is sharing the processor resources with other critical applications, it is important to avoid

penalizing the performance of these critical applications due to hardware testing. Performance penalties can be reduced by allowing the ACE testing thread to execute only when the processor resources are not utilized by the performance critical threads. An example scenario is to execute the ACE testing thread when the processor is stalled waiting for an L2 cache miss to complete, *i.e.*, in the shadow of L2 cache misses. This execution scenario is illustrated in Figure 4.5(c). In this execution model, the processor suspends the execution of the application and context switches into the ACE testing thread when the application incurs an L2 cache miss due to its oldest instruction. This context switch is similar to the lightweight context switches used in switch-on-event multithreading [2, 65]. When the L2 miss is fully serviced, the processor context switches back to the application and suspends the execution of the ACE thread. Under this execution policy, the ACE testing thread utilizes resources that would otherwise be not utilized and does not use the processor resources when these are needed by other performance critical applications. However, it is possible that the full ACE testing might not be completed in the shadow of L2 misses because the application might not incur enough L2 cache misses. If that is the case, the remaining portion of the ACE testing thread is executed at the end of the checkpoint interval.

The advantage of this ACE testing model is that it does not require a separate SMT context and can possibly provide lower performance overhead than sequential ACE testing. On the other hand, if L2 misses are not common in an application, this model can degenerate into single-threaded sequential ACE testing.

4.1.6 Flexibility of ACE Testing

The software nature of ACE testing inherently provides a more flexible solution than hardwired solutions. The major advantages offered by this flexibility are:

Dynamic tuning of the performance-reliability trade-off: The software nature of ACE testing provides the ability to dynamically trade-off performance with reliability (defect coverage). For example, when the system is running a critical application demanding high system reliability, ACE testing firmware can be run more frequently with higher quality and higher coverage targets (*i.e.*, use of different fault models and more test patterns). On the other hand, when running a performance critical application with relatively low reliability requirements (*e.g.*, video decompression), ACE testing frequency can be reduced.

Utilization-oriented testing: ACE testing allows the system to selectively test only those resources utilized by the running applications. For example, if the system is running integer-intensive applications, there might be no need to test non-utilized FPU resources.

Upgradability: Both fault models and ATPG tools are active research areas. Researchers continuously improve the quality and coverage of the generated test patterns. Therefore, during the lifetime of a processor, numerous advances will improve the quality and test coverage of the ATPG patterns. The software nature of ACE testing allows processor vendors to periodically issue ACE firmware updates that can incorporate these advances, and thus improve the defect detection quality during the processor's lifetime.

Adaptability: ACE testing allows vendors to adapt the testing method based on in-the-field analysis of likely defect scenarios. For example, if a vendor observes that the failure of a specific processor is usually originating from a particular module, they can adapt the ACE testing firmware to prioritize efforts on that particular module.

4.2 Experimental Methodology

The OpenSPARC T1 architecture, the open source version of the commercial UltraSPARC T1 (Niagara) processor from Sun [127], is used as the experimental testbed for the evaluation of the ACE framework.

The OpenSPARC T1 processor implements the 64-bit SPARC V9 architecture and targets commercial applications such as application servers and database servers. It contains eight SPARC processor cores, each with full hardware support for four threads. The eight cores are connected through a crossbar to a unified L2 cache (3MB). The chip also includes four memory controllers and a shared FPU unit [127].

First, using the processor's RTL code, the processor was divided into ACE domains. This partition was made based on functionality, where each domain comprises a basic functionality module in the RTL code. When dividing the processor into ACE domains, the modules that are dominated by SRAM structures (such as caches) were excluded because such modules are already protected with error-coding techniques such as ECC. Figure 4.6 shows the processor modules covered by the ACE framework (note that the L1 caches within each core are also excluded). Overall, the RTL implementation of the ACE framework consists of 79 ACE domains, each domain including on average 45 64-bit ACE segments. The whole chip comprises roughly 235K ACE-accessible bits.

Next, each ACE was synthesized using the Synopsys Design Compiler with the Artisan IBM 0.13um standard cell library. The test patterns were generated using the Synopsys TetraMAX ATPG tool. TetraMAX takes as input the gate-level synthesized design, a fault model, and a test coverage target and tries to generate the minimum set of test patterns that meet the test coverage target.

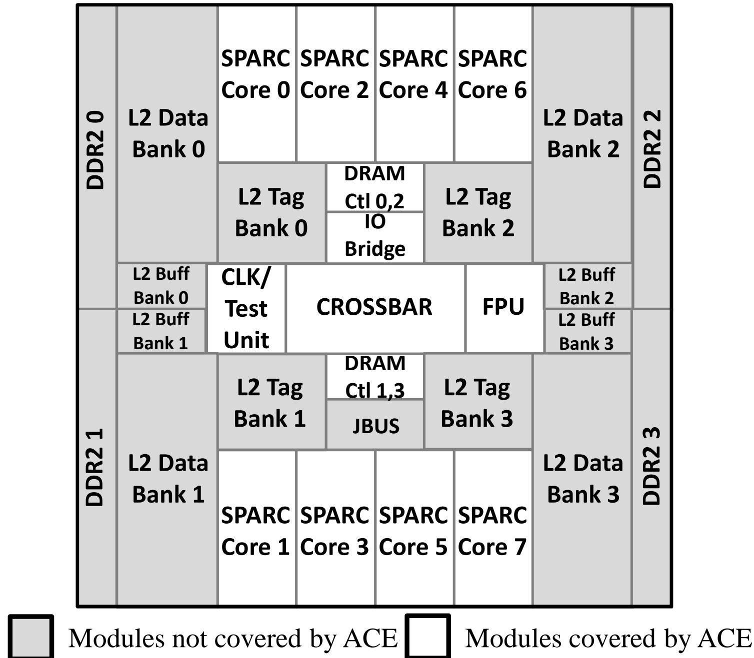


Figure 4.6: ACE Coverage of the OpenSPARC T1 Processor: Modules that are dominated by SRAM structures, such as on-chip caches, are not covered by ACE testing since they are already protected by ECC.

Fault Models: In the evaluation of the ACE framework, three single-fault models were used: stuck-at, N-detect and path-delay. The stuck-at fault model is the industry standard model for test pattern generation. It assumes that a circuit defect behaves as a node stuck at 0 or 1. However, previous research has shown that the test pattern sets generated using the N-detect fault model are more effective for both timing and hard failures, and present higher correlation to actual circuit defects [82, 40]. In the N-detect test pattern sets, each single stuck-at fault is detected by at least N different test patterns. As expected, the benefit of more effective testing by using the N-detect model comes with the overhead of larger test pattern set sizes and longer testing times. To provide the flexibility of dynamically trading off between reliability and performance, test pattern sets using both fault models were generated.

In addition to the stuck-at and N-detect fault models, test patterns were also generated using the path-delay fault model [23]. The path-delay fault model tests the design for delay faults that can cause timing violations. The test patterns generated using the path-delay fault model exercise the circuit's paths at-speed to detect whether a path is too slow due to manufacturing defects, wearout-related defects, or process variation. A detailed description of the path-delay fault model used is available in the Synopsis TetraMAX user guide [130].

Benchmarks: A set of benchmarks from the SPEC CPU2000 suite were used to evaluate the performance overhead and memory logging requirements of ACE testing.[8] All benchmarks were ran with the reference input set.

Microarchitectural Simulation: To evaluate the performance overhead of ACE testing, the SESC simulator [106] was modified to simulate a SPARC core enhanced with the ACE framework. The simulated SPARC core is a 6-stage, in-order core (with 16KB IL1 and 8KB DL1 caches) running at 1GHz [127].[9] For each simulation run, the first billion instructions were skipped and then cycle-accurate simulation for different checkpoint interval lengths (10M, 100M and 1B dynamic instructions) was performed. To obtain the number of clock cycles needed for ACE testing, a process that was emulating the ACE testing functionality was also simulated. For the SMT experiments, a separate thread that runs the ACE testing software was used with a round-robin thread fetch policy. For these experiments, the simulation terminates when the ACE thread finishes testing and at least one of the other threads executes 100M instructions.The thread combinations simulated for these experiments were determined randomly. Unless otherwise stated, the presented experimental results were obtained using the single-threaded sequential execution model of ACE testing.

Experiments to Determine Memory Logging Requirements: The Pin x86 binary instrumentation tool [77] was used to evaluate the memory logging storage requirements of coarse-grained checkpointing. A Pin tool that measures the amount of storage needed to buffer the cache lines written back from the L2 cache to main memory during a checkpoint interval, based on the ReVive checkpointing scheme [101], was implemented. Note that only the first L2 writeback to a memory address during the checkpoint interval causes the old value of the cache line to be logged in the buffer. 64 bytes (same as our cache line size) are logged for each L2 writeback. Benchmarks were run to completion for these experiments. Section 4.3.4 presents the memory logging overhead of the ACE framework.

Performance Overhead of I/O-intensive Applications: An irreversible I/O operation (*e.g.*, sending a packet to a network interface or writing to the disk) requires the termination of a checkpoint before it is executed. If such operations occur frequently, they can lead to consistently short checkpoint intervals and therefore high performance overhead for our proposal. To investigate the performance overhead due to such frequent I/O operations, some I/O-intensive file-system and network processing benchmarks were also simulated. Specifically, the microbenchmarks Bonnie and IOzone were used to exercise

[8]Results from some SPEC CPU2000 benchmarks that we were not able to port to the simulation framework are not presented.

[9]SESC provides a configuration file for the OpenSPARC T1 processor, which was used in the evaluation experiments.

the file system by performing frequent disk read/write operations. The NetPerf benchmarks [46] were also used to exercise the network interface by performing very frequent packet send/receive operations. In addition to the Netperf suite, three other benchmarks, NetIO, NetPIPE, and ttcp, that are commonly used to measure network performance were evaluated. In these experiments, the execution of an irrecoverable I/O operation is preceded by a checkpoint termination and the new checkpoint interval begins right after the execution of the I/O operation. Section 4.3.7 presents our results.

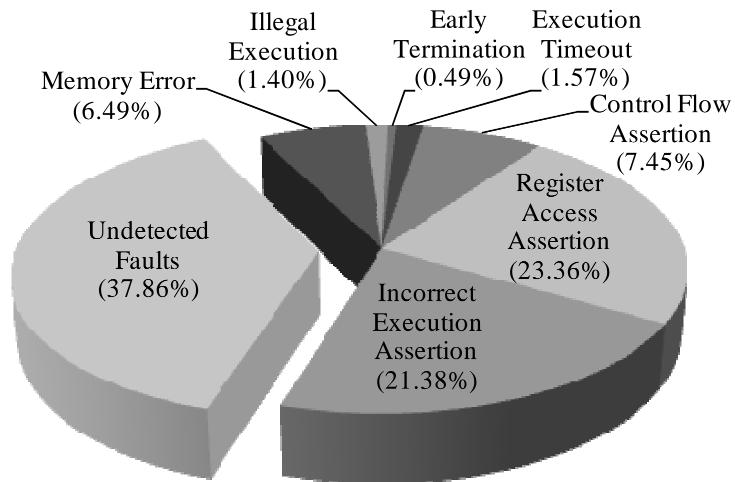
RTL Implementation: The ACE tree structure was implemented in RTL using Verilog in order to obtain a detailed and accurate estimate of the area and power consumption overheads of the ACE framework. The ACE tree design was synthesized using the same tools, cell library and methodology that was used for synthesizing the OpenSPARC T1 modules, as described earlier in this section. Section 4.3.8 evaluates and quantifies the area overhead of the ACE framework while Section 4.3.9 evaluates its power consumption.

4.3 Experimental Evaluation

4.3.1 Basic Core Functional Testing

Before running the ACE testing firmware, a software functional test is performed first to check the core for defects that would prevent the correct execution of the testing firmware. If this test does not report success in a timely manner to an independent auditor (*i.e.*, the OS running on other cores), the test is repeated to verify that the failing cause was not transient. If the test fails again, then an irrecoverable core defect is assumed, the core is disabled, and the targeted tests are canceled.

The software functional test used to check the core consists of three self-validating phases. The first phase runs a basic control flow check where 64 basic blocks are executed in a non-sequential control flow and each of the 64 basic blocks sets the value of a bit in a 64-bit architectural register. At the end of the phase, a control flow assertion checks the value of the register to determine whether or not the execution was correct. The second phase checks the core's capability to access the register file. This phase consists of a sequence of data-dependent ALU instructions that eventually read and write all architectural registers. At the end of this phase, the final result of this chain of computation is checked by an assertion. The final phase of the basic core test consists of a sequence of dependent instructions that uses each of the instructions in the ISA at least once. The final result of the functional test is checked by an assertion that validates the last generated value. The total size of the software functional test is approximately 700 dynamic instructions.



Control Flow Assertion	Incorrect execution during the control flow test.
Register Access Assertion	Incorrect execution during the register access test.
Incorrect Execution Assertion	The final result of the test is incorrect.
Early Termination	The execution terminated without executing all the instructions (wrong control flow)
Execution Timeout	The test executed for more than the required clock cycles (wrong control flow, e.g., infinite loop)
Illegal Execution	The test executed an illegal instruction (e.g., an instruction with an invalid opcode)
Memory Error	Memory request for an invalid memory address
Undetected Fault	The test executed correctly

Figure 4.7: Fault Coverage of Basic Core Functional Testing: The pie chart shows the distribution of the outcomes of a fault injection campaign on a 5-stage in-order core running the purely software-based preliminary functional tests.

A stuck-at fault injection campaign on the gate-level netlist of a synthesized 5-stage in-order core (similar to the SPARC core with the exception of multithreading support) was performed to evaluate the effectiveness of the basic core test. Figure 4.7 shows the distribution of the outcomes of the fault injection campaign. Overall, the basic core test successfully detected 62.14% of the injected faults. The remaining 37.86% of the injected faults lied in parts of the core's logic that do not affect the core's capability of executing simple programs such as the basic core test and the ACE testing firmware. ACE testing firmware will subsequently test these untested areas of the design to provide full core coverage.

These results also demonstrate that software-based functional tests that, unlike the ACE testing firmware, do not have access/control on the core's internal state, are inadequate to provide a high-quality, high-coverage test of the underlying hardware. Similar

Module	Area (mm^2)	ACE Accessible Bits	Stuck-at Test Insts	Test (%) Coverage	Path-Delay Test Insts	N-Detect Test Insts	
						N = 2	N = 4
SPARC CPU Core	8x17=136	8x19772=158176	152370	100.00	110985	234900	434382
CPU-Cache Crossbar	14.0	27645	67788	100.00	10122	117648	200664
Floating Point Unit	4.6	4620	88530	99.95	31374	126222	212160
e-Fuse Cluster	0.2	292	11460	94.70	4305	33000	68160
Clock and Test Unit	2.3	4205	68904	92.88	10626	126720	240768
I/O Bridge	4.9	10775	110274	100.00	31479	171528	316194
DRAM Controller	2x6.9=13.8	2x14201=28402	122760	91.44	126238	204312	365364
Total	175.8	234115		99.22			

Table 4.3: Test Instructions Needed to Test Each Major Modules: The table shows the number of test instructions needed by the ACE framework to test each of the major modules in the OpenSPARC T1 design.

software functional testing techniques were used for the manufacturing testing of the Intel Pentium 4 [99]. The coverage of these tests as reported in [99] is in the range of 60-70%, which corroborates the results observed from our fault-injection campaign on a simpler Niagara-based core.

4.3.2 ACE Testing Latency, Coverage, and Storage Requirements

An important metric for measuring the efficiency of the ACE framework is how long it takes to fully check the underlying hardware for defects. The latency of testing an ACE domain depends on (1) the number of ACE segments it consists of and (2) the number of test patterns that need to be applied. In this experiment, test patterns for each individual ACE domain in the design were generated using three different fault models (stuck-at, path-delay and N-detect) and the methodology described in Section 4.2. Table 4.3 lists the number of test instructions needed to test each of the major modules in the design (based on the ACE firmware code shown in Figure 4.4).

For the stuck-at fault model, the most demanding module is the SPARC core, requiring about 150K dynamic test instructions to complete the test. Modules dominated by combinational logic, such as the SPARC core, the DRAM controller, the FPU, and the I/O bridge are more demanding in terms of test instructions. On the other hand, the CPU-cache crossbar, which consists mainly of buffer queues and interconnect, requires much fewer instructions to complete the tests.

For the path-delay fault model, test pattern sets for the critical paths that are within 5% of the clock period were generated. The required number of test instructions to complete the path-delay tests is usually less than or similar to that required by the stuck-at model. Note that, with these path-delay test patterns, a defective device can cause undetected timing violations only if it is not in any of the selected critical paths and it causes extra

Design Module	Storage Requirements of Test Patterns/Responses (MB)				
	Stuck-at	Path-Delay	N-Detect (N=2)	N-Detect (N=4)	All Models
SPARC CPU Core (spac)	0.36	0.33	0.56	1.03	2.28
CPU-Cache Crossbar (ccx)	0.17	0.03	0.30	0.51	1.01
Floating-Point Unit (fpu)	0.22	0.10	0.30	0.50	1.13
e-Fuse Unit (efc)	0.03	0.01	0.08	0.16	0.27
Clock and Test Unit (ctu)	0.17	0.03	0.32	0.61	1.14
I/O Bridge (iobdg)	0.28	0.10	0.43	0.79	1.60
DRAM Controller (dram_ctl)	0.59	0.72	0.93	1.44	3.69
Total	1.83	1.34	2.91	5.04	11.11

Table 4.4: Storage Requirements for Test Patterns and Responses: Test pattern/response storage requirements per fault model and design module.

delays greater than 5% of the clock period. This probability is expected to be very low; however, stricter path selection strategies can provide higher coverage if deemed necessary (with a higher testing latency). For the specific experiments, it was found that the path selection strategy used does not lead to a large number of selected paths. However, in designs where delays of the majority of paths are within 5% of the clock period, more sophisticated path selection strategies can keep the number of selected paths low while maintaining high test coverage [94].

For the N-detect fault model, the number of test instructions is significantly more than that needed for the stuck-at model. This is because many more test patterns are needed to satisfy the N-detect requirement. For values of N higher than four, it was observed that the number of test patterns generated increases almost linearly with N, an observation that is aligned with previous studies [82, 40].

Full Test Coverage: The overall chip test coverage for the stuck-at fault model is 99.22% (shown in Table 4.3). The only modules that exhibit test coverage lower than 99.9% are the e-Fuse cluster, the clock and test unit, and the DRAM controllers, which exhibit the lowest test coverage at 91.44%. The relatively low test coverage in these modules is due to ATPG untestability of some portions of the combinational logic. In other words, no test patterns exist that can set a combinational node to a specific value (lack of controllability), or propagate a combinational node's value to an observable node (lack of observability). If necessary, a designer can eliminate this shortcoming by adding dummy intermediate state elements in the circuit to enable controllability and observability of the ATPG untestable nodes. The test coverage for the two considered N-detect fault models is slightly less than that of the stuck-at model, at 98.88% and 98.65%, respectively (not shown in Table 4.3 for simplicity).

Storage Requirements for ATPG Test Patterns/Responses: Table 4.4 shows the storage requirements for the ATPG test patterns and the associated test responses. The

Module	Cores [0,1] Test Insts		Cores [2,4] Test Insts		Cores [3,5] Test Insts		Cores [6,7] Test Insts	
	Stuck-at	Path-delay	Stuck-at	Path-delay	Stuck-at	Path-delay	Stuck-at	Path-delay
1xSPARC CPU Core	152370	110985	152370	110985	152370	110985	152370	110985
1/8xCrossbar	8474	1265	8474	1265	8474	1265	8474	1265
1/2xFPU							44265	15687
1/2xe-Fuse Cluster					5730	2153		
1/2xClock & Test Unit	34452	5313						
1/2xI/O Bridge			55137	15740				
1/2xDRAM Ctrl (pair)			61380	63119	61380	63119		
Total	195296	117563	277361	191109	227954	177522	205109	127937
Stuck-at + Path-delay	312859		468470		405476		333046	

Table 4.5: Full-Chip Distributed ACE Testing: The testing process is distributed over the chip's eight SPARC cores. Each core is assigned to test its resources and some parts of the surrounding non-core modules. The table shows the number of test instructions needed by each core pair to perform the distributed testing.

storage requirements are shown separately for each major module in the OpenSPARC T1 chip and for each fault model considered in this work. Notice that since there is resource replication in the OpenSPARC T1 chip (*e.g.*, there are eight SPARC cores and four DRAM controllers on the chip), only one set of test patterns/responses is required to be stored per resource. The least amount of test pattern storage is required by the path-delay fault model (1.34 MB) while the most demanding fault model is N-detect, where $N = 4$, which requires about 5 MB. The overall test pattern/response storage requirement for all modules and all fault models is 11.11 MB, which is similar to what is reported in previous work [74]. In ACE framework, the test patterns and responses are stored in physical memory and loaded into the register file during the testing phase. Therefore, for physical memories of several gigabytes in modern processors, the storage requirements of 11 MB is considered negligible.

4.3.3 Full-Chip Distributed Testing

In the OpenSPARC T1 architecture, the hardware testing process can be distributed over the chip's eight SPARC cores. Each core has an ACE tree that spans over the core's resources and over parts of the surrounding non-core modules (*e.g.*, the CPU-cache crossbar, the DRAM controllers *etc.*). Therefore, each core is assigned to test its resources and some parts of the surrounding non-core modules.

The testing responsibilities of the non-core modules were distributed to the eight SPARC cores based on the physical location of the modules on the chip (shown in Figure 4.6). Table 4.5 shows the resulting distribution. For example, each of the cores *zero* and *one* are responsible for testing a full SPARC core, one eighth of the CPU-cache crossbar and one half of the clock and test unit. Therefore, cores *zero* and *one* need 195K dynamic test

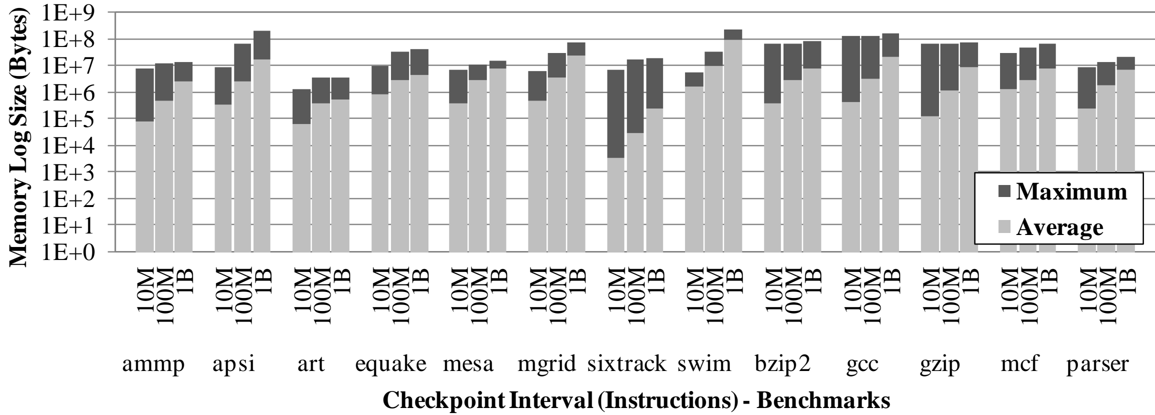


Figure 4.8: Memory Logging Storage Requirements: Average and maximum memory log size requirements for checkpoint intervals of 10 million, 100 million, and 1 billion executed instructions.

instructions to test for stuck-at faults and 117K instructions to test for path-delay faults in the parts of the chip they are responsible for. Note that the ACE tree of a core is designed such that it covers all the non-core areas that the core is responsible for testing.

The most heavily loaded pair of cores are cores *two* and *four*. Each of these two cores is responsible for testing its own resources, one eighth of the CPU-cache crossbar, one half of the DRAM controller and one half of the I/O bridge, for a total of 468K dynamic test instructions (for both stuck-at and path-delay testing). The overall latency required to complete the testing of the entire chip is driven by these 468K dynamic test instructions, since all the other cores have shorter test sequences and will therefore complete their tests sooner.

4.3.4 Memory Logging in Coarse-grained Checkpointing

The performance overhead induced by running the ACE testing firmware depends on the testing firmware's execution time and execution frequency. When ACE testing is coupled with a checkpointing and recovery mechanism, in order to reduce its execution frequency, and therefore its performance overhead, coarse-grained checkpointing intervals are required.

Figure 4.8 explores the memory logging storage requirements for such coarse-grained checkpointing intervals on the examined SPEC CPU2000 benchmarks. The memory log size requirements are shown for a system with a 2MB L2 data cache (recall that memory logging is performed only for the first L2 writeback of a cache line to main memory in a checkpoint interval [101]). For each benchmark, the average and maximum required memory log size for intervals of 10 million, 100 million, and 1 billion executed instructions are shown. The maximum metric keeps track of the maximum memory log size required

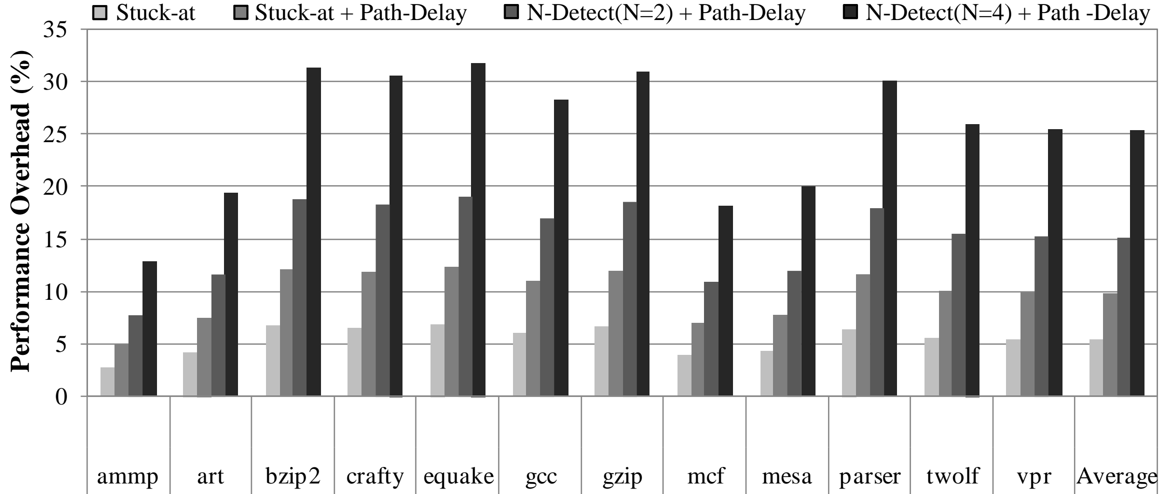


Figure 4.9: Performance Overhead of Single-Threaded Sequential ACE Testing: Performance overhead of ACE testing for a 100M-instruction checkpoint interval.

in any of the checkpoint intervals during the benchmark's execution, while the average metric averages the memory log size requirement over all the checkpoint intervals (note that the benchmarks were ran to completion with the reference inputs).

We observe that when considering checkpoint intervals that are in the order of 100 million executed instructions, the average memory log size requirements are in the range of a few kilobytes to 10MB. The most demanding benchmark is *swim*: on average it requires 1.8MB, 10MB and 91.4MB respectively for checkpoint intervals of 10M, 100M and 1B instructions. Since the memory log will be maintained at the system's physical memory, the results of this experiment suggest that checkpoint intervals of hundreds of millions of executed instructions are sustainable with insignificant memory storage overhead.[10]

4.3.5 Performance Overhead of ACE Testing

This section evaluates the performance overhead of ACE testing for the execution models described in Section 4.1.5. For all experiments, the checkpoint interval is set to 100M instructions.

Single-Threaded Sequential ACE Testing: With this execution model, at the end of each checkpoint interval normal execution is suspended and ACE testing is performed. In these experiments, the ACE testing firmware executes until it reaches the maximum test coverage. The four bars in the graph of Figure 4.9 show the performance overhead when the fault model used in ACE testing is i) stuck-at, ii) stuck-at and path-delay, iii) N-detect (N=2) and path-delay, and iv) N-detect (N=4) and path-delay.

[10]Note that most current systems are equipped with several gigabytes of physical memory.

Checkpoint Interval	Average Memory Log Size (MB)	Perf. Overhead (%) (Stuck-at)	Perf. Overhead (%) (Stuck-at + Path Delay)
10M Instr.	0.48	53.74	96.91
100M Instr.	2.59	5.46	9.85
1B Instr.	14.94	0.55	0.99

Table 4.6: Performance and Memory Log Size Tradeoffs: The Tables shows the memory log size and ACE testing performance overhead for different checkpoint intervals.

The minimum average performance overhead of ACE testing is 5.5% and is observed when only the industry-standard stuck-at fault model is used. When the stuck-at fault model is combined with the path-delay fault model to achieve higher testing quality, the average performance overhead increases to 9.8%. As expected, when test pattern sets are generated using the higher-quality N-detect fault model, the average performance overhead increases, to 15.2% and 25.4%, for N=2 and N=4 respectively.

Table 4.6 shows the trade-off between memory logging storage requirements and performance overhead for checkpoint intervals of 10M, 100M and 1B dynamic instructions. Both log size and performance overhead are averaged over all evaluated benchmarks. As the checkpoint interval size increases, the required log size increases, but the performance overhead of ACE testing decreases. This experiment demonstrates that checkpoint intervals in the order of hundreds of millions of instructions are sustainable with reasonable storage overhead, while providing an efficient substrate to perform ACE testing with low performance overhead.

SMT-Based ACE Testing: Figure 4.10 shows the performance overhead when ACE testing is used in a 2-way SMT processor with several SPEC CPU2000 benchmarks. The ACE testing thread runs concurrently, on a separate SMT context, with the benchmark that is evaluated. In this execution model, when ACE testing checks for stuck-at failures the average performance overhead is 2.6%, which is 53% lower than the 5.5% overhead observed when testing is performed in a single-threaded sequential execution environment. For other fault models, the observed results follow a similar trend: the performance overhead of SMT-based ACE testing is lower than the performance overhead of single-threaded sequential ACE testing. The performance overhead reduction observed under the SMT-based execution model stems from better processor resource utilization between the ACE testing thread and the running application. This is a consequence of the ACE testing thread simultaneously sharing the processor resources instead of sequentially executing exclusively on the processor. The latency of major portions of ACE testing (loading and checking of test patterns) is hidden by application execution.

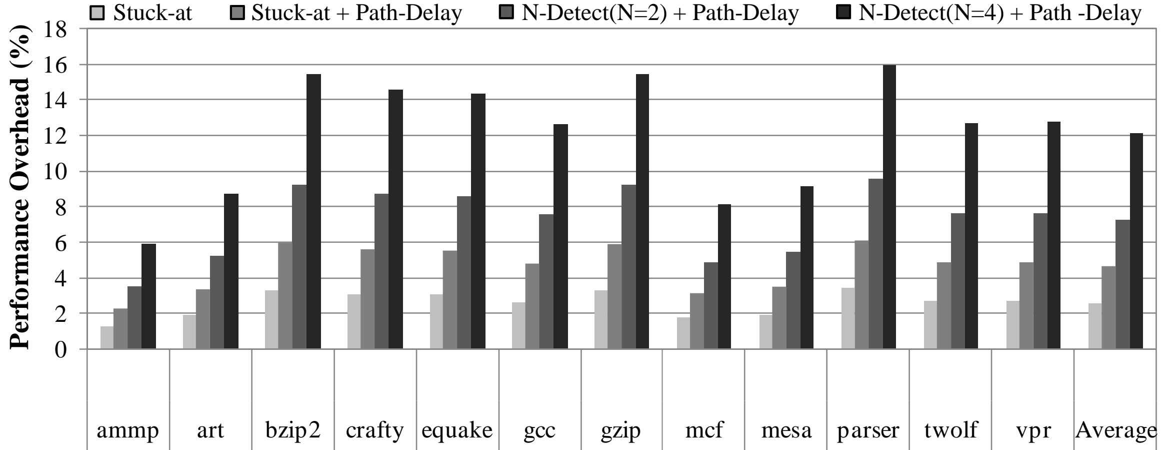


Figure 4.10: Performance Overhead of SMT-Based ACE Testing: Performance overhead of SMT-based ACE testing for a 100M-instruction checkpoint interval.

Note that, even with the SMT-based execution model, ACE testing causes performance overhead. This is due to two reasons. First, the ACE testing thread shares the processing (*e.g.*, functional units, instruction scheduler entries) and memory system resources (*e.g.*, the L1/L2 caches, buses, and DRAM memory) with the normal application thread. This resource sharing leads to interference between the two threads and delays the execution of the application thread. Second, when the ACE testing thread executes an *ACE_test* instruction, the execution of the application thread is suspended for one cycle, which also delays the application thread's execution. Even so, the contacted experiments have shown that SMT-based ACE testing results in a relatively low performance overhead for the application thread.

In SMT-based ACE testing, the testing thread occupies an SMT context. Although performing ACE-based testing in an SMT environment can reduce the potential performance overhead of testing, it is important to also evaluate the system throughput loss due to the testing thread since the extra SMT context utilized by the testing thread could otherwise be utilized by another application thread. Figure 4.11 shows the reduction in system throughput when the testing thread competes for processor resources with other threads in a 2-way and a 4-way SMT configuration. In these experiments, system throughput is defined as the number of instructions per cycle executed by application threads (excluding the testing thread). Also, for these experiments, ACE testing is performed for only one thread in the application mix, the leftmost thread for each mix shown in Figure 4.11, which is assumed to be the only application thread with high reliability requirements. We observe that, for stuck-at testing, the system throughput reduction in a 2-way SMT configuration is limited to 3%. The highest throughput reduction, 24%, is observed in a 2-way SMT configuration

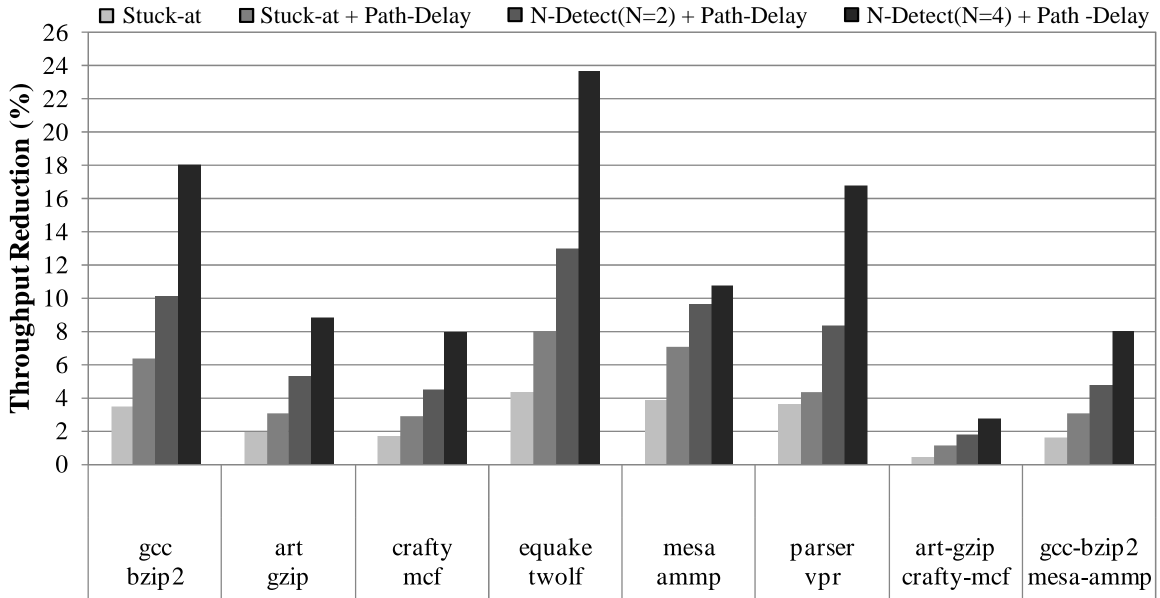


Figure 4.11: Throughput Reduction Due to SMT-Based ACE Testing: The ACE testing thread occupies an extra SMT context which leads to system throughput reduction. The graph shows the system throughput reduction due to ACE SMT-based ACE testing for a 2-way and a 4-way SMT configurations.

when high quality testing is performed (N-Detect, N=4, in combination with the path-delay fault model). We also observe that when the number of SMT contexts increases to 4, the throughput reduction due to software-based testing significantly reduces. This is because ACE testing occupies only a single thread context in the SMT processor and other thread contexts can still contribute to system throughput by executing application threads.

Interleaved ACE Testing in the Shadow of L2 Misses: Figure 4.12 shows the performance overhead when ACE testing is run in the shadow of L2 cache misses. With this execution model, whenever there is an L2 cache miss on the application thread there is a lightweight context switch with the ACE testing thread. The application thread resumes execution after the L2 cache miss is served. In the case that the checkpoint buffering resources are full (signaling the end of the checkpoint interval) and the ACE testing is not completed, the ACE testing thread starts running exclusively on the processor resources and executes the remaining of the ACE testing routine to completion. The dark part of each bar in Figure 4.12 shows the fraction of ACE testing overhead that is due to testing performed in the shadow of L2 cache misses, while the gray part shows the fraction of ACE testing overheads that is due to testing performed at the end of the checkpoint interval. The overhead of testing that is performed in the shadow of L2 cache misses is caused by the additional time taken to switch between the application thread and the ACE testing thread, and vice versa.

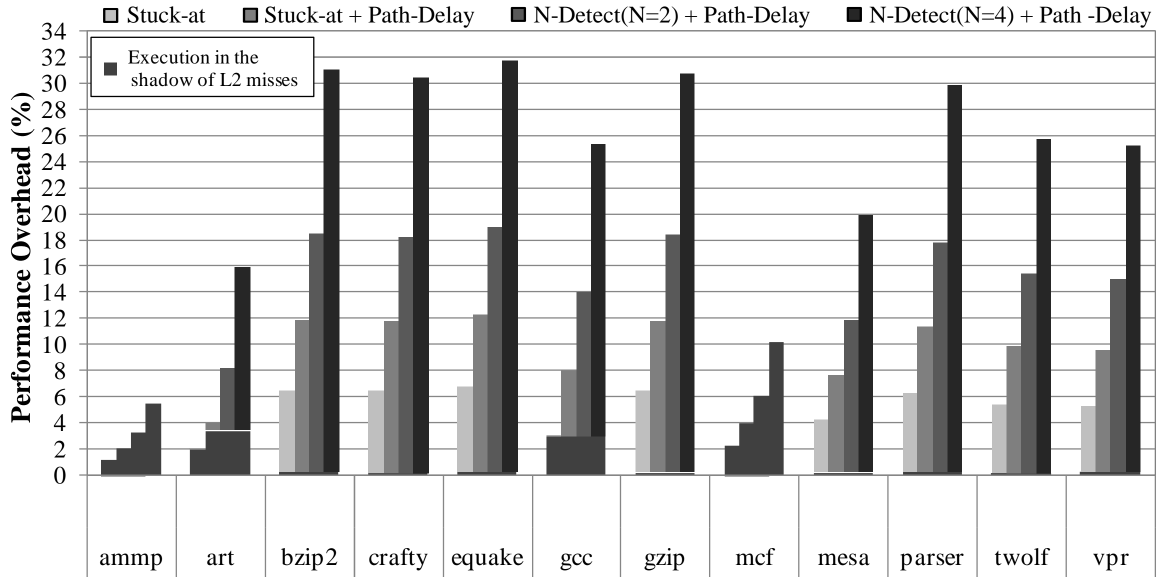


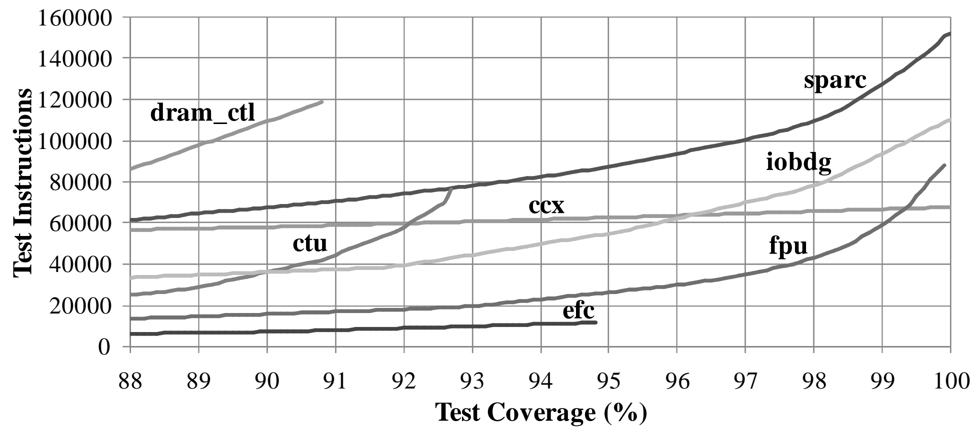
Figure 4.12: Performance Overhead of Interleaved ACE Testing: The graph shows the performance overhead of interleaved ACE testing in the shadow of L2 cache misses for a 100M-instruction checkpoint interval.

We observe that for some memory intensive benchmarks that exhibit a high L2 cache miss-rate, such as `ammp` and `mcf`, the ACE testing routine was able to run in its entirety in the shadow of L2 cache misses. For these benchmarks, we observe an average performance overhead reduction of 57% and 43% respectively compared to single-threaded sequential ACE testing. However, for the rest of the benchmarks we noticed that due to the low L2 cache miss-rate there were very few opportunities to execute the ACE testing thread in the shadow of L2 cache misses. These benchmarks, depending on the amount of ACE testing performed in the shadow of L2 cache misses, exhibit the same or slightly less performance overhead when compared to single-threaded sequential ACE testing.

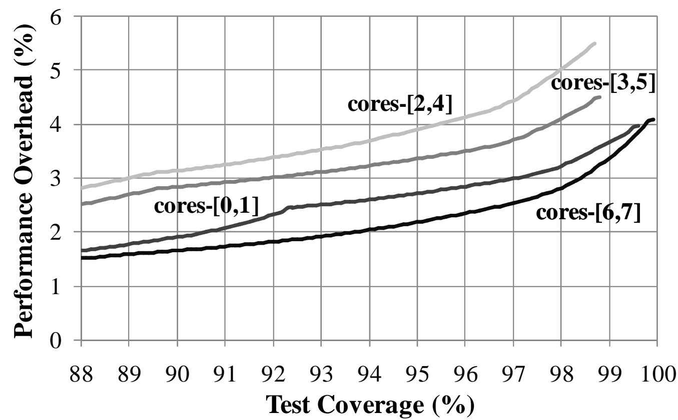
Based on these experimental results, we conclude that the interleaved ACE testing execution model benefits only benchmarks that exhibit a high enough L2 cache miss-rate and provide enough opportunities for interleaved ACE testing to utilize the processor resources more efficiently. Different thread interleaving criteria other than L2 cache misses could lead to higher benefits and affect more uniformly all benchmarks. However, the overhead of switching between the application thread and the ACE testing thread should be kept low.

4.3.6 Performance-Reliability Trade-off

The test coverage achieved by the testing firmware increases as more test instructions are executed (and therefore more test patterns are applied). However, the relation be-



(a)



(b)

Figure 4.13: Performance Overhead of ACE Testing VS. Test Coverage: Part (a) shows the number of executed test instructions versus the achieved test coverage for each of the major modules, while part (b) shows the test coverage versus performance overhead for each core pair in full-chip distributed testing.

tween the number of executed test instructions and the test coverage level is not linear. Figure 4.13(a) shows the number of executed test instructions versus the test coverage obtained for each of the major modules (using the stuck-at fault model along with the single-threaded sequential execution model for ACE testing). We observe that for some of the modules there is an exponential increase in the number of instructions needed to earn the last few percentage points of coverage. For example, the number of dynamic instructions required to achieve 100% test coverage for the SPARC core is approximately 152K, almost twice the number of instructions required to achieve 93% coverage.

This observation suggests that there is plenty of opportunity to dynamically tune the performance-reliability trade-off in the ACE testing framework. Figure 4.13(b) shows the test coverage (for the stuck-at model) versus the performance overhead for each core pair

(based on the testing partition described in Section 4.3.3). The results demonstrate that test coverage can dynamically be trade-off for reductions in the performance overhead of testing. For example, the performance overhead for cores *two* and *four* to reach 89% test coverage is only 3%. This is a 46% reduction from the performance overhead of 5.5% to reach 98.7% test coverage. This experiment demonstrates that the software-based nature of the ACE testing provides a flexible framework to trade-off between test coverage, test quality, and performance overhead.

4.3.7 Overhead of ACE Testing in I/O-intensive Applications

In I/O-intensive applications, frequent I/O operations significantly affect the performance overhead of checkpoint-based system rollback and recovery. Several system I/O operations are not reversible (*e.g.*, sending a packet to a network interface, writing to the display, or writing to the disk), and thus cause early checkpoint termination. Consequently, frequent I/O operations lead to shorter checkpoint intervals and more frequent hardware testing that can have a negative impact on system performance. This section evaluates the performance overhead of ACE testing under a heavy I/O usage environment using I/O-intensive file-system and network processing benchmarks.

Figure 4.14 shows the execution time overhead of ACE testing for the stuck-at fault model and the stuck-at combined with the path-delay fault model. Except for three of the *Netperf* benchmarks, all benchmarks exhibit an execution time overhead that ranges from 4% to 10% for the stuck at fault model and from 6% to 17% when combined with the path-delay fault model. Note that the overheads are very high (greater than 25%) in some *Netperf* benchmarks because these benchmarks are intentionally designed to stress-test the network interface, by executing a very tight loop that continuously sends and receives packets to/from the network interface. Even with these adversarial benchmarks, the performance overhead of ACE testing is at most 27% with the stuck-at fault model and 48% with the combined stuck-at and path-delay fault models.

In this experiment, a checkpoint terminates whenever there is a write operation to the file-system or a send/receive operation to the network interface (*i.e.*, an irrecoverable I/O operation). This assumption is pessimistic. The execution time overhead observed in this experiment can be significantly reduced with more aggressive and intelligent I/O handling techniques like I/O buffering [95] or I/O speculation [98], which we do not consider in these experiments. Furthermore, we note that heavily I/O-intensive applications, such as the *Netperf* benchmarks, constitute an unfavorable running environment for the ACE testing technique due to two reasons. First, if high performance is desired when running such I/O intensive applications, the system can alternatively reduce the test quality require-

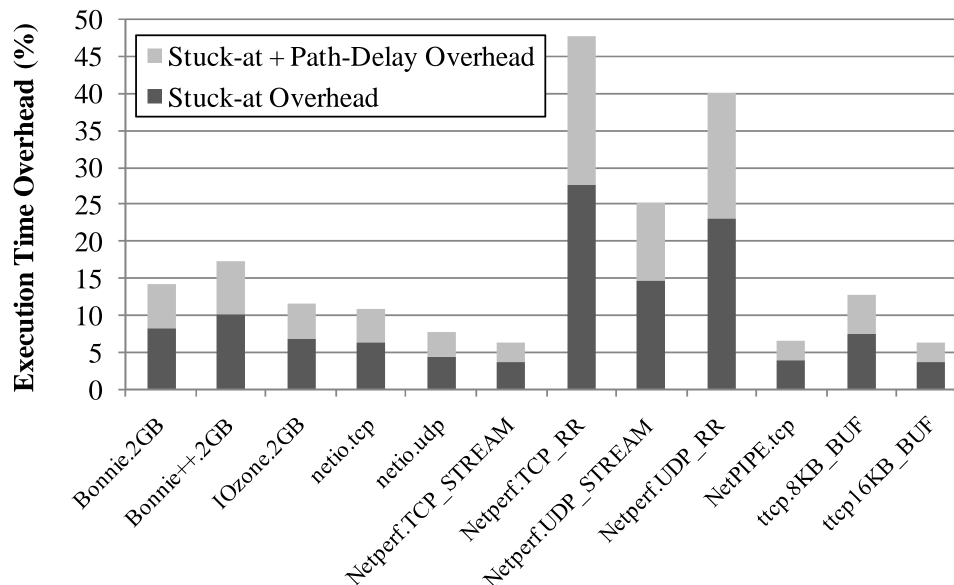


Figure 4.14: ACE Testing on I/O-Intensive Applications: The graph shows the execution time overhead of ACE testing on I/O-intensive file-system and networking applications

ments of ACE testing (or even completely switch it off) and trade-off testing quality with performance. Second, we note that such I/O intensive applications have very low CPU utilization; therefore there might be little need for high-quality, high-coverage ACE testing of the CPU during their execution.

4.3.8 ACE Tree Implementation and Area Overhead

The area overhead of the ACE framework is dominated by the ACE tree. In order to evaluate this overhead, the ACE tree for the OpenSPARC T1 architecture was implemented in Verilog and synthesized with the Synopsys Design Compiler. The ACE tree implementation consists of data movement nodes that transfer data from the tree root (the register file) to the tree leaves (ACE segments) and *vice versa*. In this specific implementation, each node has four children and therefore in an ACE tree that accesses 32K bits (about 1/8 of the OpenSPARC T1 architecture), there are 42 internal tree nodes and 128 leaf nodes, where each leaf node has four 64-bit ACE segments as children. Figure 4.15(a) shows the topology of this ACE tree configuration, which has the ability to directly access any of the 32K bits. To cover the whole OpenSPARC T1 chip with the ACE framework, eight such ACE trees were used, one for each SPARC core. The overall area overhead of this ACE framework configuration (for all eight trees) is 18.7% of the chip area.

In order to contain the area overhead of the ACE framework, a hybrid ACE tree design was implemented that combines the direct processor state accessibility of the previous

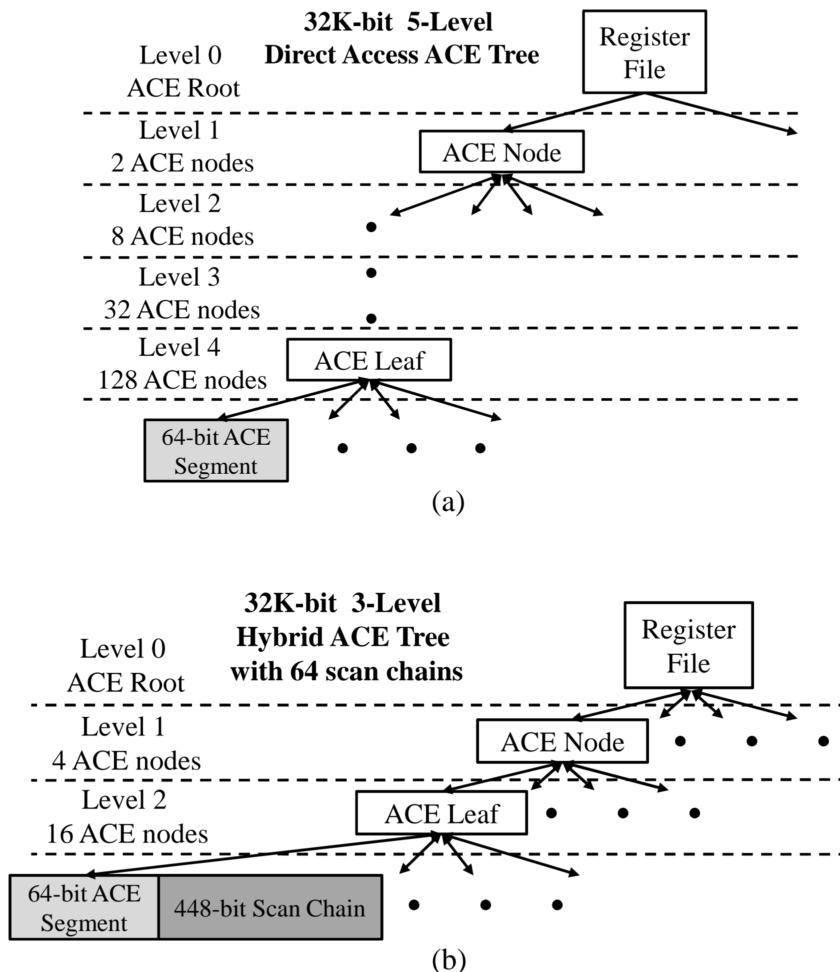


Figure 4.15: ACE Tree Implementation: Part (a) shows the topology of a direct-access ACE tree. Part (b) shows the topology of a hybrid (partial direct-access, partial scan-chain) ACE tree.

implementation with the existing scan-chain structure. In this hybrid approach, the 32K ACE-accessible bits are divided into 64 512-bit scan chains. Each scan chain has 64 bits that can be directly accessed through the ACE tree. The reading/writing to the rest of the bits in the scan chain is done by shifting the bits to/from the 64 directly accessible bits. Figure 4.15(b) shows the topology of the hybrid ACE tree configuration. The overall area overhead of the ACE framework when using the hybrid ACE tree configuration is 5.8% of the chip area.[11]

Notice that although the hybrid ACE tree is a less flexible ACE tree configuration, it does not affect the latency of the ACE testing firmware. The ACE testing firmware accesses the 64 scan chains sequentially. Since there is an interval of at least 64 cycles between two consecutive accesses to the same scan chain, data can be shifted from/to the

[11]It was found that the ACE tree's impact on the processor's clock cycle time is negligible in both direct-access and hybrid implementations.

direct access portion of the chain to/from the rest of the scan chain without producing any stall cycles. For example, during test pattern loading, each 64-bit parallel load to a scan chain is followed by 64 cycles of scan chain shifting. While the parallel loaded data is shifted into the rest of the scan chain in an ACE segment, the testing firmware loads the rest of the scan chains in the other 63 ACE segments. By the time the testing firmware loads the next 64 bits to the scan chain, the previous 64 bits have already been shifted into the scan chain. Similarly, during test response reading, each parallel 64-bit data read is followed by shifting cycles that move the next 64 bits from the scan chain to the direct access portion.

4.3.9 Power Consumption Overhead of the ACE Framework

An important consideration in evaluating the ACE framework is the degree to which the extra hardware increases the baseline design's power consumption envelope. To evaluate this power consumption overhead for our design on Sun's OpenSPARC T1 chip-multiprocessor, we first estimated the power consumption of the baseline design that lacks the ACE framework capabilities. We calibrated the estimated power consumption with actual power consumption numbers provided by Sun for each module of the chip [72]. After we validated our power estimates for the baseline OpenSPARC T1 design, we estimated the additional power required by the ACE framework.

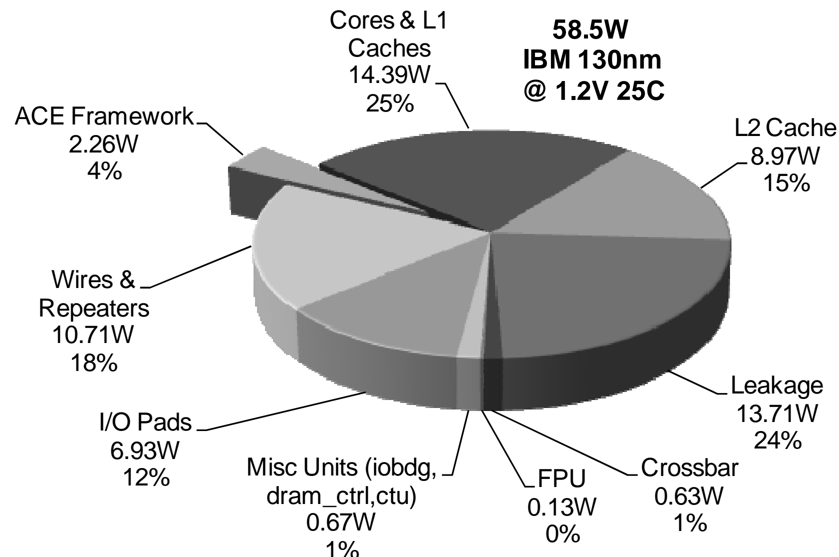
Power Estimation Methodology: Figure 4.16(a) shows the major design components of the OpenSPARC T1 and the methodology/tools we used to estimate their power consumption. We estimated the power consumption of the majority of OpenSPARC T1 modules using the Synopsys Power Compiler (part of the Synopsys Design Compiler package) and the available RTL code for the design. Each module's RTL code is synthesized using the Design Compiler. The resulting gate-level netlist is subsequently analyzed by the Power Compiler to estimate the module's power consumption. To perform the synthesis and power consumption analysis, we used the Artisan IBM 130nm standard cell library, characterized at typical conditions of 1.2V (Vdd) and 25C average temperature. The average transistor switching activity factor was set to 0.5.

For modules dominated by SRAM structures, such as the on-chip caches, where logic synthesis and power analysis using the RTL code is inefficient,[12] we used existing tools designed specifically to characterize SRAM modules. To estimate the power consumption of the L1 and L2 caches, we used the CACTI 4.2 tool [132], a tool with integrated cache performance, area, and power models.

[12]In logic synthesis memory elements are synthesized into either latches or flip-flops. Therefore, SRAM macro cells are implemented using memory compilers instead of using the conventional logic synthesis flow.

Design Component	Methodology/Tools Used
SPARC Core	Synopsys Power Compiler
L1 Inst. & Data Cache	CACTI 4.2
L2 Cache	CACTI 4.2
Crossbar	Synopsys Power Compiler
FPU	Synopsys Power Compiler
Misc. Units (iobdg, dram_ctrl,ctu)	Synopsys Power Compiler
I/O Pads	Taken from [72]
Wires & Repeaters	Taken from [72]
ACE Framework	Synopsys Power Compiler

(a)



(b)

Figure 4.16: Power Consumption Overhead of the ACE Framework: Part (a) shows all the major design components and the methodology/tools used to estimate the associated power consumption. Part (b) shows the power envelope of the OpenSPARC T1 design enhanced with the ACE framework.

This methodology is sufficient enough to estimate the power consumption of most of the chip's logic modules. However, there are parts of the design whose power consumption cannot be accurately estimated with these tools. These include 1) numerous buses, wires, and repeaters distributed all over the design, which are very hard to model accurately using the Design and Power Compilers, unless the design is fully placed and routed, 2) I/O pads of the chip. In order to estimate the power consumption of these two parts, we used values from the reported power envelope of the commercial Sun UltraSPARC T1 design [72].

Results: The estimated power envelope for the whole OpenSPARC T1 chip without the addition of the ACE framework is 56.3W.[13] Figure 4.16(b) shows the power consumption for our enhanced OpenSPARC T1 design including the ACE framework. The power envelope of the ACE-enhanced design is 58.5W, where the power consumption of the ACE framework is estimated to be 2.2W. Thus, the ACE framework consumes 4% of the design's total power. Our estimation assumes that the ACE framework is enabled all the time while the chip is in operation. However, as illustrated in the previous sections, the ACE framework is actually used during very short testing periods at the end of each checkpoint interval. Therefore, we expect the actual power consumption and power envelope overhead of the ACE framework to be significantly lower than 4%, depending on the frequency and length of testing (*i.e.*, checkpoint interval size and time spent in testing).

4.4 Related Work

Hardware-Based Reliability Techniques: A traditional defect detection technique that is predominantly used for manufacturing testing is logic BIST [23]. Logic BIST incorporates pseudo-random pattern generation and response validation circuitry on the chip. Although on-chip pseudo-random pattern generation removes any need for pattern storage, such designs require a large number of random patterns and often provide lower fault coverage than ATPG patterns [23].

The ACE framework improves on this traditional defect detection technique due to the following major reasons: 1) it effectively removes the need for on-chip test pattern generation and validation circuitry and moves this functionality to software, 2) it is not hardwired in the design and therefore has ample flexibility to be modified/upgraded in the field (as described in Section 4.1.6), and 3) it has higher test coverage and shorter testing time because it uses ATPG instead of pseudo-randomly generated patterns.

A more recent work, CASP [74], proposes the use of ATPG test patterns, stored in non-volatile memory (*e.g.*, hard disk), as a system self-test technique that runs concurrently with normal operation. Hardware testing in CASP is orchestrated by an on-chip hardware controller. To initiate the CASP self-test process, the controller suspends normal execution on a core and isolates the core from the rest of the cores by disabling its interconnect links with other cores. This enables the application of test patterns to exercise and test the core's integrity while preserving correctness. The hardware controller loads the test patterns from non-volatile storage into the serial scan chain of the core, observes

[13] Our estimate of the OpenSPARC T1 power is within 12% of the reported power consumption of the commercial Sun Niagara design [72].

the scan response generated by the core, and checks if the response matches the correct response that is also stored in non-volatile storage. After the self-test process is finished, normal execution resumes on the core. In [74], CASP is evaluated only for testing single cores and it does not cover non-core modules.[14] The major differences between CASP and the ACE framework are: 1) in CASP the self-test process is orchestrated by an on-chip test controller whereas ACE framework exposes the microarchitecture to software so that software/firmware test programs can perform self test, 2) CASP loads the test patterns through the slow, serial scan chain structure whereas in ACE testing test patterns are loaded into the scan state through the very fast, parallel-loadable ACE infrastructure. As a result, ACE testing results in a lower area/power overhead (no need for an on-chip test controller) and orders of magnitude faster testing time (CASP testing time is in the order of seconds [74]. In contrast, the testing time for the ACE framework is in the order of milliseconds).

Smolens *et al.* [119] proposed a detection technique for emerging wearout defects that periodically runs functional tests that check the hardware under reduced frequency guardbands. Their technique leverages the existing scan chain hardware for generating hashed signatures of the processor's microarchitectural state summarizing the hardware's response to periodic functional tests. This technique allows the software to observe a signature of the microarchitectural state, but it does not allow the software to directly control (*i.e.*, modify) the microarchitectural state. In contrast, the ACE framework approach provides the software with direct and fast *access and control* of the scan state using the ACE infrastructure. This direct access and control capability allows the software to run online directed hardware tests on any part of the microarchitectural state using high-quality test vectors (as opposed to functional tests that do not directly control the microarchitectural state and do not adhere to any fault model). Furthermore, the proposed direct fast access to the scan state enables the validation of each test response separately (instead of hashing and validating all the test responses together), thereby providing finer-grained defect diagnosis capabilities and higher flexibility for dynamic tuning between performance overhead (*i.e.*, test length) and test coverage.

Finally, in Table 4.7, we compare the ACE framework mechanism [28] to other defect-tolerance solutions, both traditional techniques and techniques that were proposed more recently in the research literature. Table 4.7 is an updated version of Table 3.4 presented in Section 3.4. The research-stage solutions presented in Table 4.7 are listed in chronological order with the less recent at the top. A detailed description of these techniques is provided in Section 3.4. From Table 4.7 we observe that among the research-stage solutions, the ACE framework provides the highest defect coverage (99%), with the lowest area over-

[14] However, CASP can be extended to test non-core modules.

Defect Tolerance Solution	Defect Coverage	Area Overhead	Performance Overhead	Design Intrusion/ Complexity	Comments
Traditional Solutions					
Dual Modular Redundancy (DMR)	Very High (~99%)	Very High (>100%)	Very Low (<5%)	Low	Provides only error detection. Easy to cover the whole design.
Triple Modular Redundancy (TMR)	Very High (~99%)	Ultra High (>200%)	Very Low (<5%)	Low	Provides both error detection and forward recovery. Easy to cover the whole design.
N-Version Redundancy	Very High (~99%)	Very High (>100%)	Very Low (<5%)	Very High	*N* different versions of the component have to be implemented.
Error Correction Codes (ECC)	Memory Structures	Medium (~15%)	Very Low (<5%)	Low	Limited only to memory structures or data buses.
Research-Stage Solutions					
DIVA Austin [6]	Not Available	Low (~6%)	Not Available	Medium	Uses an online checker at the pipeline's retirement stage.
SRAS Bower *et al.* [19]	Only Array Structures	Not Available	Not Available	Medium	Limited to array structures. Requires hardware changes in the array structures.
Bower *et al.* [20]	Not Available	Medium (>15%)	Not Available	High	Uses DIVA checkers and pipeline additions that truck instruction execution for defect diagnosis.
BulletProof [116, 83]	High (~95%)	Medium (~14%)	Ultra Low (<1%)	Medium	Uses BIST-like on-chip hardware checkers.
ElastIC Sylvester *et al.* [129]	Under Development/Evaluation			High	Uses on-chip sensors, silicon wear-out prediction units, and on-chip testers.
Argus Meixner *et al.* [85]	High (~98%)	Medium (~11%)	Low (~4%)	Medium	Uses runtime checkers for the validation of control flow, computation, dataflow, and memory operations.
ACE Framework [28]	Very High (~99%)	Low (~6%)	Low (~5%)	Low	Add architectural support for ACE-based testing: ACE Tree (CAD tools) + ISA Extensions
StageNet Gupta *et al.* [42]	Not Available	Medium (~15%)	Medium (~10%)	High	Pipeline stages need to be isolated and connected through crossbar switches. No error detection support.

Table 4.7: Comparing The ACE Framework To Related Work: Comparison of the ACE Framework to traditional defect-tolerance solutions and more recent techniques found in the research literature. The techniques are compared in respect to their defect coverage, area overhead, performance overhead, and the degree they intrude in the original design and they are presented in chronological order with the less recent at the top.

head (6%), for a very low runtime performance overhead (5%). When compared to the traditional defect-tolerance solutions, the ACE framework provides the same defect coverage as the traditional techniques, but at a much lower area overhead. The only drawback of ACE framework when compared to traditional defect-tolerance solutions is the higher degree of intrusion in the original design and its higher design complexity.

Software-based Reliability Techniques: A very recent approach proposes the detection of silicon defects by employing low overhead detection strategies that monitor for simple software symptoms at the operating system level [73]. These software-based

detection techniques rely on the premise that silicon defects manifested in some microarchitectural structures have a high probability (∼95%) to propagate detectable symptoms through the software stack to the operating system [73].

The main differences between [73] and the ACE framework are: 1) unlike the probabilistic software symptom-based defect detection, the ACE framework checks the underlying hardware in a deterministic process through a structured high-quality test methodology with very high fault coverage (99%) and can be executed on demand, 2) software symptom-based defect detection techniques can flag the possible existence of a hardware failure, but they do not have the capability to diagnose which part of the underline hardware is defective. In ACE framework, by employing ATPG test patterns, it is trivial to diagnose the defective device at a very fine granularity.

There are numerous previous works, such as [105, 113, 13], that proposed the use of software-based techniques for online detection of soft errors. However, none of them addresses the problem of online defect detection.

Instruction-based Functional Testing: A large amount of work has been performed in functional testing [21, 63, 70] of microprocessors. The most relevant of these to the ACE framework are the instruction-based functional self-test techniques. In general, these techniques apply randomly-generated or automatically-selected instruction sequences and/or combinations of instruction sequences and randomly- or automatically-generated operands to test for hardware defects. If the result of the test sequence does not match the expected output of the instruction sequence, then a hardware fault is declared. We briefly describe the state-of-the-art approaches that work in this manner. In [133], a self-test program written in processor assembly language and the expected results of the program are stored in on-chip ROM memory. When invoked, the self-test program performs at-speed functional testing of the processor. The proposed scheme requires very little additional hardware cost. It requires an LFSR for generating randomized operands for test instructions and a MISR for generating the result signature. Also, a minor modification of the ISA is required for the test instructions to read/write from the LFSR/MISR. Similarly, [66] uses the knowledge of the ISA and the RTL-level model of a processor to select high fault-coverage instructions and their operands to include in self-test software routines. Batcher and Papachristou [11] employ instruction randomization hardware to generate randomized instructions to be used in self-test software routines for functional testing. Brahme and Abraham [21] describe how to generate randomized instruction sequences to be used in self-test software routines. Building upon these works, Chen and Dey [26] propose a mechanism that generates instruction sequences to exercise structural test patterns designed to test processor components and applies such instruction sequences in the software-based self-test rou-

tines to achieve higher coverage than other approaches that randomly generate instruction sequences.

The ACE framework is fundamentally different from these instruction-based functional testing techniques in that it is a structural testing approach that uses software routines to apply test patterns. We introduce new instructions that are capable of applying high-quality ATPG-generated structural test patterns to every processor segment by exposing the scan chain to the instruction set architecture. Software self-test routines that use these instructions can therefore directly apply test patterns to processor structures and read test responses, which results in the fast and high-coverage structural testing of each processor component. In contrast, none of the previously-proposed instruction-based functional testing techniques are capable of directly applying test patterns to processor components. Instead, they execute existing ISA instruction sequences to indirectly (functionally) test the hardware for faults. As such, previous instruction-based functional test approaches in general lead to higher testing times or lower fault coverage since they rely on (randomized) functional testing.

One recent previous work [99], employed purely software-based functional testing techniques during the manufacturing testing of the Intel Pentium 4 processor (see Section 4.1.2 for a discussion of this work). In the ACE framework, we use a similar functional testing technique (the "basic core functional test" program) to check the basic core functionality before running the ACE firmware to perform directed, high-quality testing. In fact, any of the previously proposed instruction-based functional testing approaches can be used as the basic core functional test within the ACE framework.

Checkpointing Mechanisms: There is also a large body of work proposing various versions of checkpointing and recovery techniques [120, 101, 95]. Specifically, SafetyNet [120] provides a unified, lightweight checkpoint/recovery mechanism. Conceptually, the SafetyNet mechanism maintains multiple system-wide, consistent checkpoints of the state of a shared memory multiprocessor. After a detected fault, SafetyNet is capable of recovering the processor state to a pre-fault, error-free checkpoint. To enable system recovery, SafetyNet adds on-chip checkpoint buffers to log the memory and architectural changes across checkpointing intervals. Therefore, the size of checkpoint intervals in SafetyNet is limited by the size of the on-chip log buffers.

To address this limitation and offer longer checkpoint intervals, ReViVE [101] proposes the use of main memory to maintain the checkpoint logs. The drawback of this approach is that due to the memory-based checkpoint logging, ReViVE can cause more network and memory traffic that might result to larger performance overheads than SafetyNet.

Another limitation of both ReViVe and SafetyNet is the handling of I/O operations. In both schemes, system recovery cannot undo/redo I/O operations and therefore checkpoints cannot cross I/O operation. This limitation results to either delaying all I/O operations until the end of the current checkpoint interval, which leads to a significant performance overhead, or to the termination of the current checkpoint whenever an I/O operation occurs. In the latter case, applications with frequent I/O operations can cause shorter checkpoint intervals that in our mechanism can result to more frequent ACE-based hardware testing and higher runtime performance overhead. To address this limitation, ReViVeI/O [95] proposes a checkpoint and recovery mechanism based on ReViVe but with the additional capability of undoing and redoing I/O operations, thus enabling checkpoint intervals that can cross I/O operations. Another work that addresses the recovery of I/O operations in checkpoint and recovery mechanisms is [98] that proposes the implementation of speculative I/O operations at the operating system level.

Although no real system today employs similar checkpointing and recovery techniques, the simulation-based results from these works conclude that coarse-grained checkpoint intervals are feasible for complex commercial designs. The coarse-grained checkpoint/recovery substrate provided by such techniques enables efficient ACE testing with low performance overhead.

4.5 Chapter Summary

This chapter introduced a novel, flexible software-based technique, ISA extensions, and microarchitecture support to detect and diagnose hardware defects during online operation of a chip-multiprocessor. The technique uses the Access-Control Extension (ACE) framework that allows special ISA instructions to access and control virtually any part of the processor's internal state. Based on this framework, special firmware periodically suspends the processor's execution and performs high-quality testing of the underlying hardware to detect defects. Several execution models for the interaction of the special testing firmware with the applications running on the processor were described, for both single-threaded and simultaneously-multithreaded processing cores.

Using a commercial ATPG tool and three different fault models, the ACE framework was experimentally evaluated on a commercial chip-multiprocessor design based on Sun's Niagara. The experimental results showed that ACE testing is capable of performing high-quality hardware testing for 99.22% of the chip area. Based on a detailed RTL implementation, implementing the ACE framework requires a 5.8% increase in Sun Niagara's chip area and a 4% increase in its power consumption envelope.

Finally, it was demonstrated how ACE testing can be seamlessly coupled with a coarse-grained checkpointing and recovery mechanism to provide a complete defect-tolerance solution. The evaluation showed that, with coarse-grained checkpoint intervals, the average performance overhead of ACE testing is only 5.5%. The results also showed that the software-based nature of ACE testing provides ample flexibility to dynamically tune the performance-reliability trade-off at runtime based on system requirements.

CHAPTER V

ACE Framework Extensions - Adding Value to Resiliency Mechanisms

The previous chapters introduced the BulletProof approach and the ACE framework that, compared to traditional techniques, provide a very low cost defect-tolerance solution for microprocessor designs. Although the hardware overhead of these techniques is estimated to be around 5-10% of the chip's area, today, the designers of mainstream microprocessors still consider this hardware cost high to be dedicated solely for defect tolerance. Instead, due to the very low failure rate of current silicon process technologies, microprocessor designers prefer to use that part of the microprocessor's hardware budget for performance improvement modules like bigger memory caches and on-chip memory controllers, and limit defect-tolerance mechanisms only to the most unreliable microprocessor components like memory caches, in the form of error correction codes (ECC).

However, as devices scale into smaller sizes and the failure rate of future silicon process technologies is rising, at some point, even mainstream microprocessor designs will require to protect the whole processor design with defect-tolerance mechanisms in order to provide adequate reliability standards to the user. This transition, can be made smoother and easier to adopt if the hardware resources used for defect-tolerance mechanism could also be used for other important applications (*i.e.*, adding value to the resiliency mechanisms).

This chapter, describes how the ACE framework defect-tolerance mechanism can be extended in such a way that its hardware resources can be used by three other applications. Specifically, Section 5.1 describes how the ACE framework can be extended to provide online design bug detection and Section 5.2 compares the proposed mechanism with previous research approaches. Next, Section 5.3 describes how the ACE framework can be extended to improve two important phases of the microprocessor design cycle; the post-silicon debugging process and the manufacturing testing. Finally, the chapter is summarized in Section 5.4.

5.1 ACE Framework for Online Design Bug Detection

This Section, describes how the ACE framework hardware can be extended to perform online design bug detection. First, Section 5.1.1 provides a brief overview of the problem of design bugs in microprocessor designs and motivates the need of online design bug detection mechanisms for future generation microprocessors. Next, Section 5.1.2 highlights previous design bug analysis studies and discusses their shortcomings, followed by a rigorous RTL level design bug analysis on a commercial chip-multiprocessor. Based on the insights drawn from the RTL design bug analysis, Section 5.1.3 describes how design bugs can be detected at runtime while the microprocessor is operating at the customer side. Section 5.1.4 demonstrates how this online design bug detection technique can be implemented by extending the ACE framework hardware. The proposed mechanism is experimentally evaluated in Section 5.1.6.

5.1.1 The Problem of Design Bugs in Modern Microprocessors

The Challenges of Correct Design - The advent of chip-multiprocessing has led to unprecedented levels of chip integration. Today, most general purpose processor chips are equipped with multiple cores, multiple levels of coherent memory, on-chip interconnection networks, and memory and I/O controllers. At the same time, processors are augmented with new technologies such as virtualization, dynamic power management, and 64-bit extensions. Complex interactions between these modules, as well as the complexity of the modules themselves, put a tremendous pressure in the verification of the system. Although the verification phase of modern processors can consume a large portion of the design cycle [9], require significant amount of resources [39], and utilize state-of-the-art verification techniques, *design bugs* (also known as errata, design defects, or design errors) still slip into the final products and *"buggy"* processors find their way into the field.

This trend is clearly shown in Figure 5.1. We studied the errata documentation of five recent Intel processors and found that the rate of design bugs discovered after product release has more than doubled in the latest generation of processors.[1] The graph shows the number of discovered design bugs over the lifetime of five Intel processors. The Pentium 4, Pentium M, and the Xeon 1.4-3.2 processors exhibit a similar trend with an average of 1.2 design bugs discovered per month during their lifetime.[2] On the other hand, the higher chip-level integration of resources and the addition of new features in the Core Duo

[1] The data is extracted from the processors' errata documentation [55, 54, 51, 53, 52].

[2] We suspect that the reason why the Pentium M processor had less design bugs than the other two processors is because it was based on the matured Intel P6 architecture.

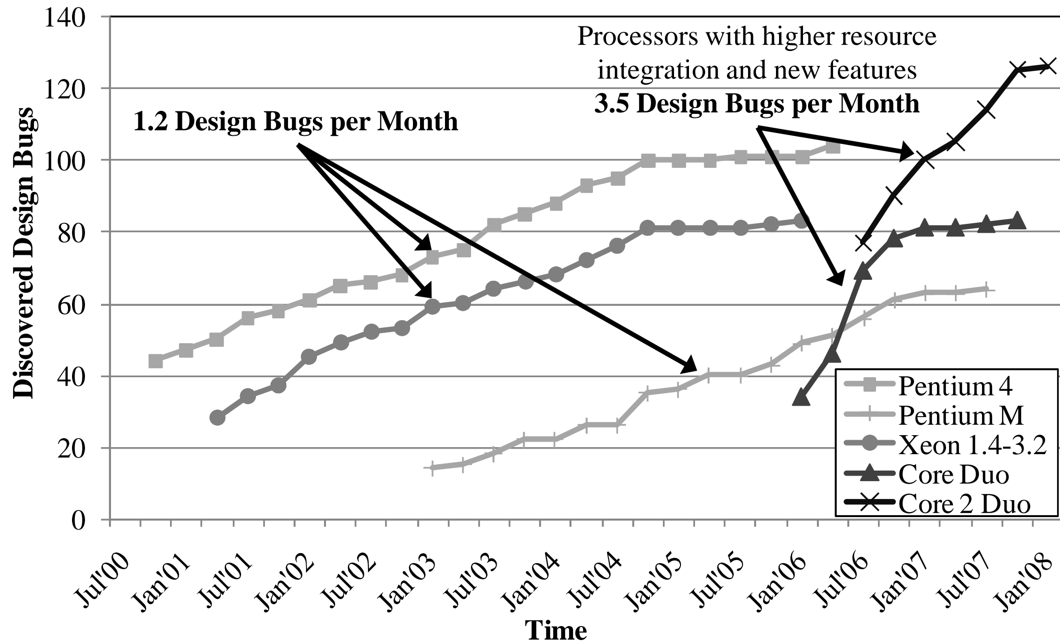


Figure 5.1: Design Bugs in Modern Microprocessors: Timeline of discovered design bugs over the lifetime of five Intel processors.

and Core 2 Duo processors resulted in more design bugs. For example, although the Core Duo dual-core processor was derived from the Pentium M single-core processor and had the same architecture, it exhibited a much higher rate of design bugs than its predecessor. Specifically, the design bug discovery rate of the two multi-core processors is 3.5 design bugs per month, almost triple that of their single-core predecessors. This trend is expected to worsen in the future as technology scaling will allow for more diverse resources to be integrated into a single chip.

Why is Online Bug Detection Needed? Today, design bugs are treated with *ad-hoc* heuristic techniques that seek to avoid the occurrence of design bugs through software and hardware configuration changes [78]. A common approach employed by such techniques to avoid the occurrence of design bugs is disabling some processor features that trigger the design bugs (*e.g.*, support for cache prefetching [78], dynamic power management [1], *etc.*). However, this often leads to reduced product quality/performance and lower customer satisfaction. Furthermore, when such workarounds are not possible, design bugs can lead to expensive product recalls [145] and a potentially diminishing brand/company reputation.

Augmenting a design with a mechanism that enables a systematic approach to detect and avoid design bugs after the product release and while the system is operating in the field can offer the following benefits:

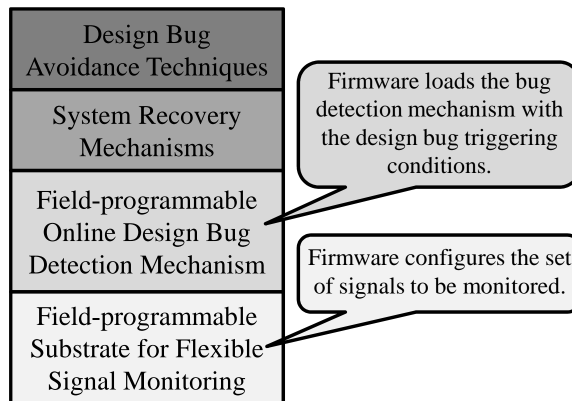


Figure 5.2: Overview of Online Design Bug Detection and Avoidance: An in-the-field design bug and avoidance framework requires 1) a flexible field-programmable substrate for signal monitoring, 2) a field-programmable design bug detection mechanism, 3) a system recovery mechanism, and 4) design bug avoidance techniques. The ACE framework hardware will be extended to provide the first two layers.

1. Faster design cycle and time to market. Today, a significant fraction of the verification phase is spent to discover a very small number of design bugs [38]. This time can be saved by discovering and fixing that small number of design bugs in the field after product release.

2. Reduce the risk of expensive product recalls (and potentially damaged company reputation) due to *ad-hoc* heuristic techniques that might not be able to avoid a discovered design bug. A systematic online design bug detection technique increases the probability of successfully dealing with the design bug and avoiding expensive recalls.

3. Avoid potential impact to product quality and customer satisfaction due to the use of conventional techniques that disable design features to avoid design bugs. Instead, online design bug detection allows the system to operate with all its features enabled and recover the system only when the design bug occurs. Therefore, during bug-free execution the system is operating under its original specifications.

Online Design Bug Detection and Avoidance - A high-level overview of an online design bug detection and avoidance framework is shown in Figure 5.2. The framework has four layers: 1) The bottom layer that provides a field-programmable substrate for flexible signal monitoring. This substrate is programmed by special firmware at system startup to select the set of signals that are required to be monitored for design bug detection. 2) A field-programmable design bug detection mechanism that checks if the monitored signals match with a bug triggering condition. The mechanism is programmed by special firmware at system startup with the bug triggering conditions. 3) A system recovery mechanism

that rolls back the system state to the last correct state when a design bug occurrence is detected. 4) Design bug avoidance techniques that are activated after a design bug detection to guide execution around the bug triggering conditions and avert the design bug. This chapter, will demonstrate how the ACE framework hardware can be extended to provide the first two layers of the online design bug detection mechanism.

5.1.2 Design Bug Analysis

We first analyze design bugs in a real processor to obtain insights into their characteristics and to develop a mechanism that can flexibly and efficiently detect the occurrence of design bugs while the system is in operation.

Previous Design Bug Analysis Studies

The potential of augmenting future microprocessors with online design bug detection has led to a number of studies that analyzed the known design bugs that slipped into recent commercial microprocessors. The objective of these studies was to better understand and gain insights into the characteristics of the known design bugs in existing microprocessors, and extrapolate the expected characteristics of the design bugs of future microprocessors.

Specifically, Avžienis *et al*. [8] analyzed the known design bugs in the Intel Pentium II since its initial release. More recently, Sarangi *et al*. [110] analyzed the design bugs in ten modern commercial microprocessors from Intel, AMD, IBM and Motorola, and Narayanasamy *et al*. [96] analyzed the design bugs in two microprocessors: Intel's Pentium 4 and AMD's Athlon 64. Another study by Wagner *et al*. [141] analyzed the design bugs in Intel StrongARM SA1100 and IBM PowerPC 750GX. The analysis in all of these studies was based on information extracted from the available microprocessor errata sheets *e.g.* [56, 1, 36]. An errata sheet is a document published and maintained by the microprocessor manufacturer to provide its customers with details about known microprocessor design bugs. The document provides an assessment of each design bug's severity, the degree to which it can affect the system, a possible set of conditions that can trigger the design bug, any possible workarounds, and sometimes the company's intention to provide a fix in a future version of the microprocessor.

A major drawback of using the errata sheets to extrapolate statistics about design bugs is that the errata sheets commonly provide very high-level descriptions of the design bugs. Such descriptions provide little or no insight into the low-level details of the underlying hardware problem. An example description of a design bug listed in the Intel Pentium 4 errata sheet [56] is shown in Figure 5.3(a). This design bug is related to complex interac-

```
R31. Interactions between the Instruction Translation Lookaside
Buffer (ITLB) and the Instruction Streaming Buffer May Cause
Unpredictable Software Behavior

Problem: Complex interactions within the instruction fetch/decode
unit may make it possible for the processor to execute instructions
from an internal streaming buffer containing stale or incorrect
information.

Implication: When this erratum occurs, an incorrect instruction
stream may be executed resulting in unpredictable software behavior.
```

(a)

```
63 - TLB Flush Filter Causes Coherency Problem in Multiprocessor
Systems

Description: If the TLB flush filter is enabled in a multiprocessor
configuration, coherency problems may arise between the page tables
in memory and the translations stored in the on-chip TLBs. This can
result in the possible use of stale translations even after software
has performed a TLB flush.

Potential Effect on System: Unpredictable system failure.
```

(b)

Figure 5.3: Design Bugs Documented in Microprocessor Errata Sheets: Examples of design bugs from (a) the Pentium 4 errata sheet, and (b) the Opteron errata sheet.

tions between the processor's instruction translation lookaside buffer and the instruction streaming buffer that can result in the execution of an incorrect instruction stream with unpredictable software behavior. Using this description, it is very hard to accurately relate this design bug to the actual hardware implementation and reason about, for example, exactly what hardware signals (*i.e.*, wires) need to be monitored by an online design bug detection mechanism to effectively detect the occurrence of the design bug. Figure 5.3(b) shows another example design bug description, from AMD's Opteron errata sheet [1]. This bug is related to the translation lookaside buffer flush filter and can lead to unpredictable system behavior. Again, from this high-level description, it is very difficult to infer the set of hardware signals that should be examined to dynamically detect its occurrence. Without knowing the set of hardware signals that needed to be monitored to detect the bug, it is very difficult to design a mechanism that would detect the bug and to accurately estimate the hardware cost of such a mechanism.

In order to design a hardware mechanism that detects design bugs, the signals that affect the occurrence of each bug need to be known. Our goal in this section is to perform a more rigorous, lower-level (RTL) analysis of design bugs. Our purpose is to understand design bug characteristics at the register transfer level to (1) design a flexible mechanism

that can detect known design bugs during online operation after the chip is manufactured, and (2) more accurately estimate the hardware cost of such a design bug detection mechanism. To this end, we first draw insights from our analysis of design bugs found and fixed in an existing commercial processor, Sun's OpenSPARC T1.

RTL Design Bug Analysis

In this Section, we perform a design bug analysis at the Register Transfer Level (RTL) in an attempt to bridge the gap between the high-level design bug descriptions provided by the microprocessor errata sheets and the low-level hardware implementation details needed to devise effective online design bug detection mechanisms. At the RTL level, the microprocessor design behavior is described in a hardware description language (*e.g.*, Verilog or VHDL). This level is considered to be very close to the actual hardware implementation. The only design phases separating the RTL level with the actual hardware implementation are 1) logic synthesis, which generates the design's gate-level netlist and 2) place-and-route, which creates the transistor-level layout of the netlist. Therefore, the direct relation between the RTL level and the underlying implementation provides an adequate level of detail that allows the extraction of low-level design bug characteristics.

Our study focuses on the Verilog RTL source code of the OpenSPARC T1 chip-multiprocessor [127], the open source version of Sun's commercial UltraSPARC T1 (Niagara) chip-multiprocessor. Since no errata documentation is publicly available for the UltraSPARC T1 microprocessor, we focus on the actual design bugs found during the development of the OpenSPARC T1 and documented in the RTL source code. Specifically, when the designers corrected a design bug, they left the original buggy code in the RTL source file as a comment. Therefore, both the original erroneous implementation as well as the fixed implementation are available in the source code. As such, by examining these two implementations, it is straightforward to discover what hardware signals are involved in each design bug. Although these design bugs did not slip into the final product, we believe they share similar characteristics with the design bugs that eventually slipped into the released version of the microprocessor with the exception of some differences which we discuss in the next section.

Methodology: We analyzed 296 design bugs that were documented in the Verilog source files of two OpenSPARC core units. These bugs account for about 99% of all documented and commented-out bugs in the OpenSPARC T1 RTL. We classified these bugs into three major classes: 1) *Logic* design bugs, 2) *Algorithmic* design bugs, and 3) *Timing* design bugs. Later, in Section 5.1.3, we analyze the logic signals that need to be monitored to detect these bugs.

```
Example 1 from Verilog file tlu_tcl.v

line 1089:   assign  intrpt_taken =
line 1090:              rstint_taken | hwint_taken | sirint_taken;
...
line 1105:   // modified for bug 3919
line 1106:   // assign       trap_to_redmode = trp_lvl_at_maxtlless1 &
             ~intrpt_taken;
line 1107:   assign  trap_to_redmode = trp_lvl_at_maxtlless1 & ~(rstint_taken
             | sirint_taken);
```


Figure 5.4: Logic Design Bug: Example of a logic design bug at the RTL level.

```
Example from Verilog file lsu_qctl1.v

line 2993:   //bug4814 - change rrobin_picker1 to rrobin_picker2
line 2993:   // Choose one among 4 loads.
line 2994:   //lsu_rrobin_picker1 ld4_rrobin  (
line 2995:   //    .events             ({ld3_pcx_rq_vld,ld2_pcx_rq_vld,
line 2996:   //                  ld1_pcx_rq_vld,ld0_pcx_rq_vld}),
...
line 3007:   //    .se(se),
line 3008:   //    .so()
line 3009:   //  );
line 3010:
line 3011:   lsu_rrobin_picker2 ld4_rrobin  (
line 3012:        .events              ({ld3_pcx_rq_vld,ld2_pcx_rq_vld,
line 3013:                         ld1_pcx_rq_vld,ld0_pcx_rq_vld}),
...
line 3020:    .se(se),
line 3021:    .so()
line 3022:  );
```


Figure 5.5: Algorithmic Design Bug: Example of an algorithmic design bug at the RTL level.

Classification of Design Bugs

Logic Design Bugs: This class of design bugs is characterized by erroneous logic in combinational circuits. A logic bug occurs because the designer formed an erroneous logic block; for example an AND gate could be used instead of an OR gate, or an inverted signal rather than the non-inverted one. The code segment presented in Figure 5.4, taken from the OpenSPARC T1 Verilog source files, illustrates an example of a logic design bug. The design bug is located in the core's trap logic unit (TLU) and is associated with the combinational logic that computes the control signal `trap_to_redmode`. The incorrect combinational circuit implementation is commented out in line 1106. The corrected combinational circuit implementation is shown in line 1107. By examining lines 1089-1090, we notice that the signal replaced in the correct code (`intrpt_taken`) is computed by ORing three other signals. One of the three signals (`hwint_taken`) is no longer a source signal in the correct implementation. We observed that many logic design bugs cannot be

```
Example from Verilog file lsu_qdp1.v
                                                           Correct Code
line 1228:   // Begin - Bug3487.
...
line 1239:   dff  #(48) ifu_std_d1 (
line 1240:           .din    (tlb_st_data[47:0]),
line 1241:           .q      (lsu_ifu_stxa_data[47:0]),
line 1242:           .clk    (asi_data_clk),
line 1243:           .se     (1'b0),     .si (),          .so ()
line 1244:           );
line 1245:                                                          Buggy Code
line 1246:   // select is now a stage earlier, which should be
line 1247:   // fine as selects stay constant.
line 1248:   //assign  lsu_ifu_stxa_data[47:0] = tlb_st_data_d1[47:0] ;
line 1249:
line 1250:   // End - Bug3487.
```

Figure 5.6: Timing Design Bug: Example of a timing design bug at the RTL level.

fixed by simply redefining the logic between the source signals in the buggy implementation. Instead, it is very common that fixing the bug requires the addition or removal of signals to/from the buggy implementation (more than 95% of logic design bugs had this requirement).

This example demonstrates the amount of low-level information provided in the RTL code that is missing from the design bug descriptions in the errata documentation. For instance, by observing the code segment associated with the design bug, it is very easy to find the set of hardware signals that activate the bug (*i.e.*, `trp_lvl_at_maxtlless1`, `rstint_taken`, `hwint_taken`, and `sirint_taken`). *In analyses solely based on errata sheets, this low-level information is abstracted away in the high-level design bug description and has to be inferred, a process that involves a high amount of uncertainty and inaccuracy.*

Algorithmic Design Bugs: This class covers major design bugs related to the algorithmic implementation of the design. These design bugs exhibit algorithmic deviations from the design specification and they usually require major modifications to be fixed. Figure 5.5 illustrates an example algorithmic design bug located in the load queue control logic at the core's load/store unit. This bug is due to an incorrect implementation of the round robin algorithm for selecting one of the four loads buffered in the load queue. To fix the incorrect round robin implementation described in module `lsu_rrobin_picker1`, a new module had to be implemented (`lsu_rrobin_picker2`). Unlike fixes for logic design bugs, fixes for algorithmic design bugs are not limited to combinational circuit modifications, rather they sometimes require multiple major modifications that can span the whole module.

Timing Design Bugs: This third class of design bugs is associated with the timing correctness of the implementation. We have observed that most of these design bugs are

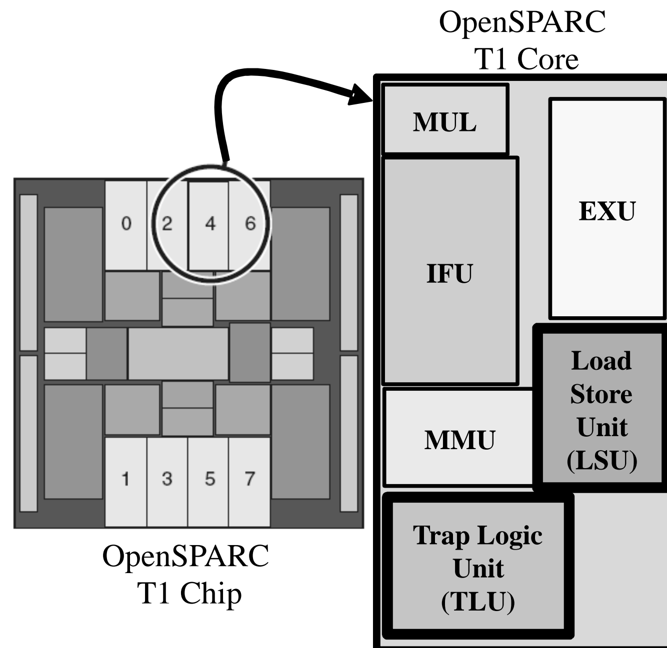


Figure 5.7: Design Bugs in the OpenSPARC T1 Core: After studying the OpenSPARC T1 Verilog source files we found that almost all of the documented design bugs are located in two units, the load/store unit (LSU) and the trap logic unit (TLU).

cases where a signal needed to be latched a cycle earlier or a cycle later in order to keep the timing of signals correct in the design. An example of such a design bug is shown in Figure 5.6. This timing design bug is located in the queue data path of the core's load/store unit. As shown in the Verilog source code, the incorrect implementation in line 1248 assigns the value of the 48-bit `tlb_st_data_d1` bus to the `lsu_ifu_stxa_data` bus in the same cycle. However, as shown in lines 1239-1244, the correct timing of the data movement between the two buses requires the data to be latched for one clock cycle. We found that the most common fix for this class of design bugs is the addition or removal of flip-flops to adhere to the timing constraints required to keep the design correct.

Design Bug Type Distribution

After studying the OpenSPARC T1 Verilog source files [126] we found that almost all (∼99%) of the documented design bugs are located in two units, the load/store unit (LSU) and the trap logic unit (TLU) [127], shown in Figure 5.7. The LSU processes all data memory access instructions. It interfaces with all the functional units and it serves as the gateway between the SPARC core and the core-cache crossbar to the memory subsystem. The LSU also includes the core's data TLB and L1 cache. The TLU implements the SPARC core's trap and software interrupt handling logic. It supports six trap levels ranging from hypervisor and supervisor mode traps to user mode traps and is capable of handling

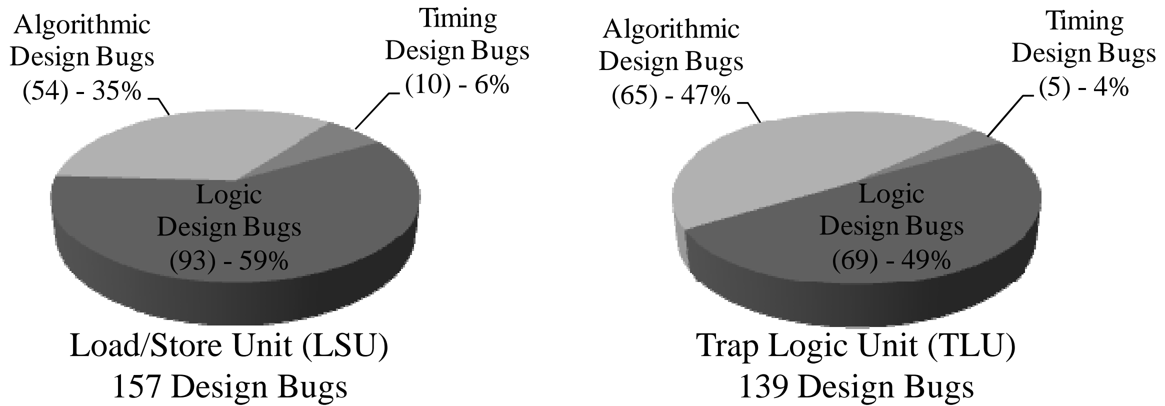


Figure 5.8: Design Bug Distribution: The graphs show the design bug distribution for the Load/Store Unit (LSU) and the Trap Logic Unit (TLU).

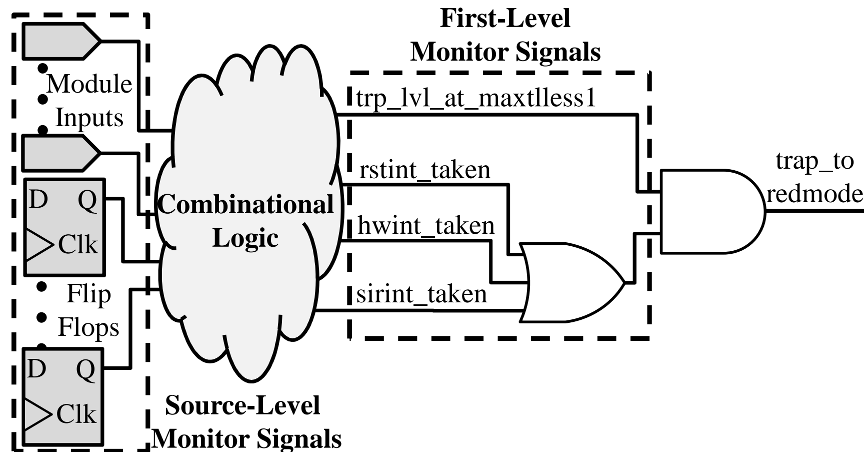
up to 64 pending software interrupts per thread. In our study we analyzed a total of 296 design bugs documented in these two units.

Figure 5.8 shows the design bug type distribution. A large fraction of the documented design bugs in the two units belong to the logic design bug class, which accounts for 59% and 49% of the total design bugs for the LSU and the TLU, respectively. The second most frequent design bug class is algorithmic design bugs, while timing design bugs are less frequent and account for only ∼5% of all bugs. The dominance of logic design bugs over the other two bug classes might imply that the process of implementing complex combinational logic is more prone to human error than implementing the algorithmic or timing specifications of the design.

As mentioned earlier in this section, these design bugs were discovered, fixed, and documented before the final tape-out of the design. As such, we expect them to have some differences with the design bugs that escape the verification phase and slip into the final product. We suspect that the algorithmic and timing design bugs have a more severe impact on the design's correctness and therefore they might have a higher probability of being discovered during the design verification phase. In contrast, because logic design bugs are isolated and localized to small combinational logic portions, they could be less likely to be discovered during the verification of the chip. This is because the erroneous effects of the logic design bugs either might not be exercised or might be masked before propagating to observable outputs during testing. For example, in order for the logic bug illustrated in Figure 5.4 to be active, the source combinational circuit must be set to specific values (which might be a rare combination of values). Based on this reasoning, the distribution of design bugs that actually slip into the final product might have fewer algorithmic and timing design bugs than the distribution shown in Figure 5.8.

Input Signals	trp_lvl_at_maxtlless1	0	0	0	0	0	0	0	0	1	1	**1**	1	1	1	1	1
	rstint_taken	0	0	0	0	1	1	1	1	0	0	**0**	0	1	1	1	1
	hwint_taken	0	0	1	1	0	0	1	1	0	0	**1**	1	0	0	1	1
	sirint_taken	0	1	0	1	0	1	0	1	0	1	**0**	1	0	1	0	1
Output Signal	trap_to_redmode (buggy logic)	0	0	0	0	0	0	0	0	1	0	**0**	0	0	0	0	0
	trap_to_redmode (correct logic)	0	0	0	0	0	0	0	0	1	0	**1**	0	0	0	0	0

(a)



(b)

Figure 5.9: Design Bug Triggering and Source Signals: In part (a), the shaded column shows the values that source signals need to take to trigger the logic bug shown in Figure 5.4. Part (b) shows the source-level and first-level signals for the same logic bug.

5.1.3 Detecting Logic Design Bugs at Runtime

Although logic design bugs might be harder to discover than the other two design bug classes, we believe that once they have been discovered, it is much easier to detect their occurrence while the "buggy" microprocessor is in operation in the field. Their characteristic of being isolated in a combinational logic circuit portion makes it possible to deterministically detect their occurrence by monitoring the values of their source signals. To illustrate this concept, we consider the logic bug example shown in Figure 5.4. By computing the truth table of the buggy circuit (line 1106) and the correct circuit (line 1107), as shown in Figure 5.9(a), we can infer that the design bug occurs when the source signals are set to a specific combination of values (shown in the shaded column of the table). Therefore, by monitoring the values of the bug's source signals it is possible to deterministically detect the occurrence of the specific design bug. In this work, we call this set of signals *first-level monitor signals* (*i.e.*, signals that directly determine the occurrence of the design bug).

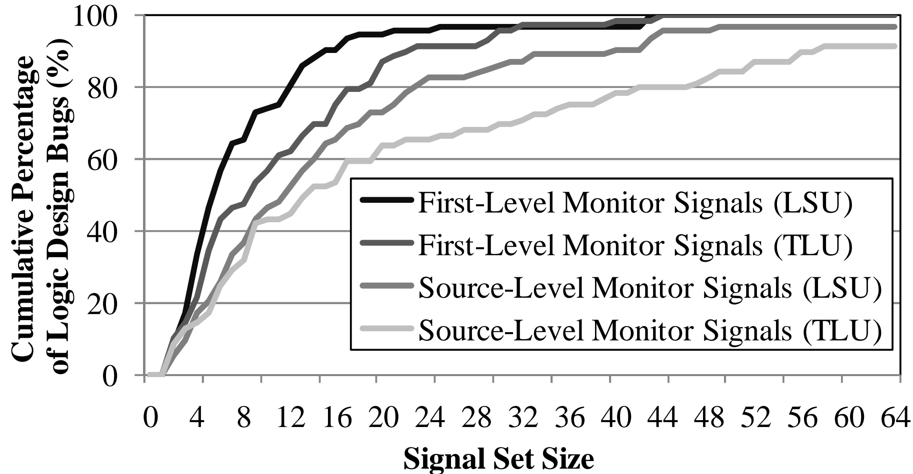


Figure 5.10: Design Bugs Source Signal Statistics: Cumulative distribution of logic bugs versus the first-level and source-level monitor signal set sizes for the LSU and TLU.

For this specific bug, the size of the first-level monitor signal set is 4 because there are 4 signals whose values directly determine the bug's occurrence.

Although it is easy to find the set of first-level monitor signals in the RTL model, these signals unfortunately might not exist in the lower transistor-level implementation due to the logic synthesis process and optimizations employed during the process of translating the RTL implementation to gate-level and then to transistor-level implementation [131]. Thus, there is not a guaranteed one-to-one mapping between signals in the RTL and signals in the transistor-level implementations. However, the logic synthesis process maintains a one-to-one mapping of the state-holding elements (*e.g.*, flip-flops) and module-level primary inputs/outputs[3] between the RTL and transistor-level implementations [131]. To effectively detect the occurrence of a logic design bug in the transistor-level hardware implementation, we need only to trace back the combinational logic that feeds the first-level monitor signals to a set of signals that are directly connected to either 1) state-holding elements or 2) primary inputs of the module. We call this set of signals the *source-level monitor signals*. Figure 5.9(b) illustrates this process. Monitoring the source-level monitor signal set of a design bug allows the detection of the bug. Note that it is simple to construct a truth table using the source-level monitor signals instead of the first-level monitor signals to understand which combination of the values assigned to source-level signals would exercise the bug.

To determine the number of signals required to be monitored to detect the occurrence of logic design bugs, we measured the first-level and source-level signals of the 162 logic design bugs located in the LSU and the TLU units. Figure 5.10 shows the cumulative

[3]In this work we consider a module to be a Verilog design module in the RTL code.

Metrics	LSU	TLU
Min./Average/Max. number of first-level monitor signals per logic design bug	2/8/43	2/12/44
Min./Average/Max. number of source-level monitor signals per logic design bug	2/17/97	2/24/89
Source-level monitor signal sharing among different design bugs	68%	64%
Average number of unique source-level monitor signals per logic design bug	6	9
Unique source-level monitor signals (for all logic design bugs)	516	602

Table 5.1: Logic Design Bug Statistics: The table lists several statistics regarding the first-level and source-level design bug signals.

distribution of the logic design bugs versus the first-level and source-level monitor signal set sizes in the LSU and the TLU units. We observe that 97% of the logic bugs located in the LSU and 92% of those located in the TLU have a source-level monitor signal set size that is smaller than 64 signals. This means that for detecting any *single* bug that is within the aforementioned percentage, at most 64 signals need to be monitored.

Table 5.1 focuses on the number of first-level and source-level signals needed to be monitored to detect logic design bugs. An interesting observation is that the average set size of source-level monitor signals per logic bug is about double the size of the first-level monitor signal set. Notice that the size of the first-level monitor signal set determines the minimum number of RTL signals required to be monitored to precisely detect the occurrence of a certain bug, given that those signals exist in the actual hardware implementation, and can be probed. On the other hand, the size of the source-level monitor signal set determines the number of transistor-level signals required to be tapped to detect a bug, given that design flip-flops and module inputs can be probed. Furthermore, the average number of source-level monitor signals per logic design bug is 17 and 24 for the LSU and the TLU units respectively (The minimum and maximum set sizes are presented as well). Hence, the detection of an average design bug requires 17 to 24 transistor-level signals to be monitored.

The total amount of tapped signals can be small if there is a high degree of source signal sharing between multiple design bugs. To quantify this, we studied the amount of sharing between the 162 logic bugs covered by our study. We found that the sharing between the source-level monitor signal sets is about 65% on average (68% in LSU and 64% in TLU). This means that 65% of the signals that belong to the source-level monitor

signal set of a logic design bug also belong to the source-level monitor signal set of at least one other logic design bug. Furthermore, each logic design bug has on average 6-9 signals in its source-level monitor signal set that are unique, *i.e.*, they do not belong to the source-level monitor signal set of any other logic design bug. This result implies that the discovery of a new design bug requires the monitoring of an additional 6-9 signals, on average, that have not been previously monitored for any other bug, thus increasing the total number of tapped signals.

In order to detect all the 162 studied logic design bugs, 516 and 602 unique source-level monitor signals need to be monitored in the LSU and the TLU modules, respectively. Note that these numbers are much higher than previous work estimates that used high-level errata documentation to analyze design bugs. Specifically, the study in [110] reports that on average, for the ten processors studied, only 210 signals need to be monitored to detect all design bugs in all modules of a processor, with the maximum requirement out of the ten microprocessors being 260 signals. The study in [96] reports that monitoring only 41 signals is adequate to detect the occurrence of 43 out of the 63 known design bugs in the AMD Athlon 64 and AMD Opteron microprocessors. In contrast, our study shows that 1118 signals need to be monitored to detect 162 bugs in two modules of the SPARC core. We believe this discrepancy stems from the attempt in previous studies to infer low-level hardware implementation information from the high-level, abstract information provided in the microprocessor errata documents. By studying the documented design bugs at the lower RTL level, we found that the signal monitoring requirements of online design bug detection are significantly higher than the estimates of these previous studies. As a result, the problem of detecting design bugs is more difficult and the solution is likely more hardware intensive than estimated by previous work.

Insights from RTL Design Bug Analysis

In summary, our RTL design bug analysis provides the following conclusions and insights:

1. The design bugs documented in the Verilog source files of the OpenSPARC T1 chip-multiprocessor can be classified into three major classes based on their characteristics: logic, algorithmic, and timing design bugs (Section 5.1.2).

2. Logic design bugs outnumber the documented design bugs of the other two design bug classes. Furthermore, they might dominate the distribution of design bugs that escape the verification phase and slip into the final product (Section 5.1.2).

3. Because they only affect combinational logic, the occurrence of logic design bugs is more readily detectable while the system is in operation. This can be done deterministically by monitoring a set of source-level signals.

4. The number of signals that need to be monitored to detect the occurrence of logic design bugs is significantly higher than estimates provided by previous work. The discovery of a new design bug requires the monitoring of additional 6-9 signals, on average, that have not been previously monitored for any other bug.

These conclusions and insights call for a mechanism capable of concurrently monitoring a large number of different signals scattered in the design and thus providing an effective and efficient substrate to perform online detection of the occurrence of logic design bugs. In the rest of this section we describe how the ACE framework can be extended to provide such a mechanism.

5.1.4 ACE-Based Distributed Online Bug Detection

Figure 5.11 illustrates the high-level architecture of the ACE-based online design bug detection mechanism. The mechanism is characterized by two phases: 1) the initial setup of the mechanism and 2) the cycle-by-cycle operation where design bugs are detected while the system is operating in the field.

Initial Setup Process

The first step of the mechanism's setup process is the determination of the triggering conditions for each design bug in the system. The design bug triggering conditions are characterized by (1) the bug's source-level monitor signals and (2) their values that would activate the bug. The design bug triggering conditions of each bug are determined by system engineers after performing the bug analysis process presented in Section 5.1.3.

Bug Signatures: Once bug triggering conditions are determined, they are represented by a structure called a *bug signature* (step 1 in Fig. 5.11). Conceptually, the bug signature is a list of all the signals in the system. From that list, the bug's source-level monitor signals are marked with the value they need to take to trigger the bug, while non-source signals are marked with a `don't care` value (X) indicating that their values are irrelevant to the bug activation. The bug signature can be considered as a representation of the system state that would activate the design bug. Each design bug can have multiple bug signatures due to multiple possible combinations of triggering conditions.

System Bug Signature: The next step in setting up the design bug detection mechanism is the generation of the *system bug signature*. The collection of bug signatures of

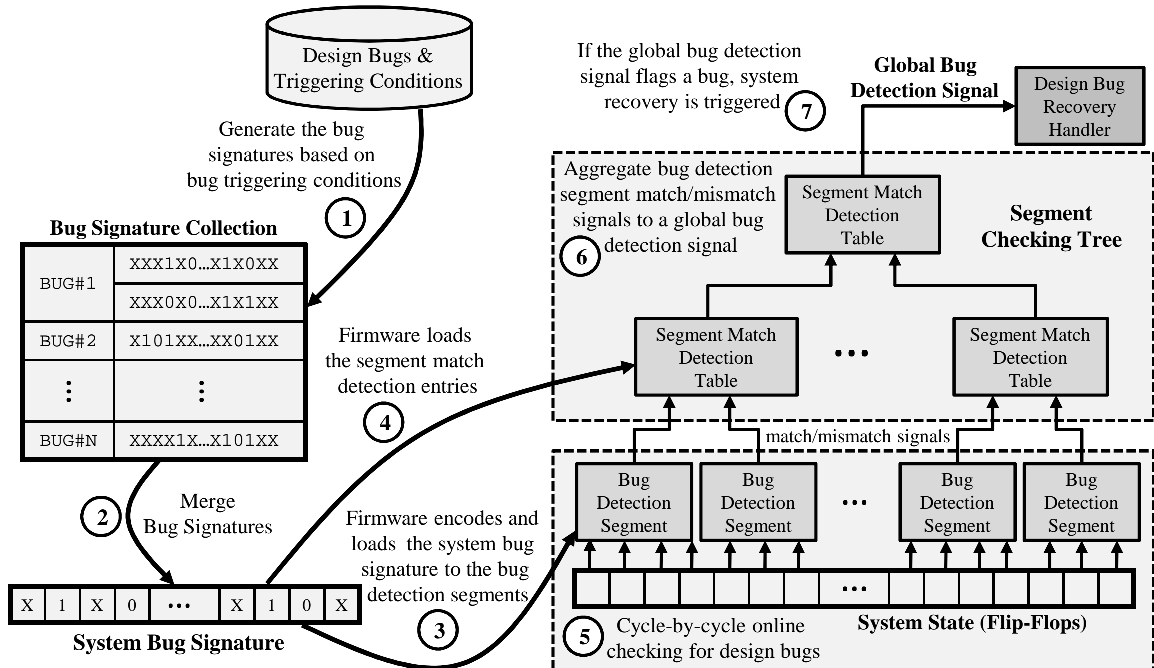


Figure 5.11: Overview of ACE-Based Online Design Bug Detection: The ACE-Based online design bug detection is partitioned in the initial setup phase (steps 1-4) and the online bug detection phase (steps 5-7).

all design bugs are merged together to form the *system bug signature* (step 2 in Fig. 5.11). The system bug signature constitutes a representation of all the conditions that can trigger any individual design bug in the system. The process of merging multiple bug signatures into the system bug signature is detailed in Section 5.1.4.

Bug Detection Segments: The system bug signature is subsequently encoded into a binary representation, partitioned into segments, and loaded into the mechanism's *bug detection segments* (step 3 in Fig. 5.11). The bug detection segments are field programmable structures each associated with a part of the system state (*i.e.*, the system's flip-flops). Each bug detection segment is loaded with the part of the system bug signature corresponding to its part of the system state. The loading of the bug detection segments is done by firmware that has access to the segments' field programmable resources. During system operation, the bug detection segments compare the system state to the system bug signature and generate match/mismatch signals.

Segment Match Detection Tables: The source-level signals of a design bug might be located only in some of the bug detection segments. Therefore, each bug is associated with a *segment match detection entry* that indicates which lower-level segments need to match the system bug signature with the system state for the bug to be detected. In essence, the system bug signature summarizes all the triggering conditions from all bugs whereas each segment match detection entry demultiplexes them to enable the detection of individual

bugs. The segment match detection entries are loaded into the segment match detection tables by firmware (step 4 in Fig. 5.11).

Cycle-by-cycle Operation and Design Bug Detection

Flip-Flop Level Checking: Once the initial setup of the mechanism is done by the firmware, the remaining task of the mechanism is to check if the system steps into a bug triggering state while it is operating. To check this, each bug detection segment compares its portion of the system bug signature to the system state and generates a segment-wide match/mismatch signal (step 5 in Fig. 5.11).

Segment Checking Tree: The detection of each individual bug usually requires only a subset of all the bug detection segments in the design to match their portion of system bug signature with the system state. For each bug, this information is encoded into a segment match detection entry. However, the set of segments that are required for the detection of an individual bug might be scattered in different areas of the chip. To aggregate the match/mismatch signals of all the segments on the chip, our mechanism employs a distributed *segment checking tree*. The structure of the checking tree is identical to the ACE tree presented in the previous chapter, only with some minor modifications. Specifically, each node in the segment checking tree has a *segment match detection table* that is populated with the segment match detection entries of each bug that has bug-detection required segments connected to the tree node. These entries are loaded during the initial setup phase by firmware (step 4 in Fig. 5.11). During system operation, if the match/mismatch signals of the underlying segments match with one of the node's segment match detection entries, this indicates that the local triggering conditions of a design bug within that node are met. In a similar fashion, each level of nodes in the tree generates a match/mismatch signal and feeds the upper level of nodes (step 6 in Fig. 5.11). If a match signal propagates all the way from the bug detection segment level to the top level of the tree, this indicates that the triggering conditions of a specific design bug are met for the whole chip and a *global bug detection signal* is asserted. This process is illustrated in detail with an example in Section 5.1.5. The bug is subsequently flagged to the *bug recovery handler* (step 7 in Fig 5.11).

Design Bug Recovery Handler: If a bug is flagged by the global bug detection signal, the design bug recovery handler recovers the system into the last validated system state. Execution is then restarted and guided by design bug avoidance techniques so that the design bug is averted, if possible. Since our focus is on bug detection, we leave the design of the bug recovery handler to future work.

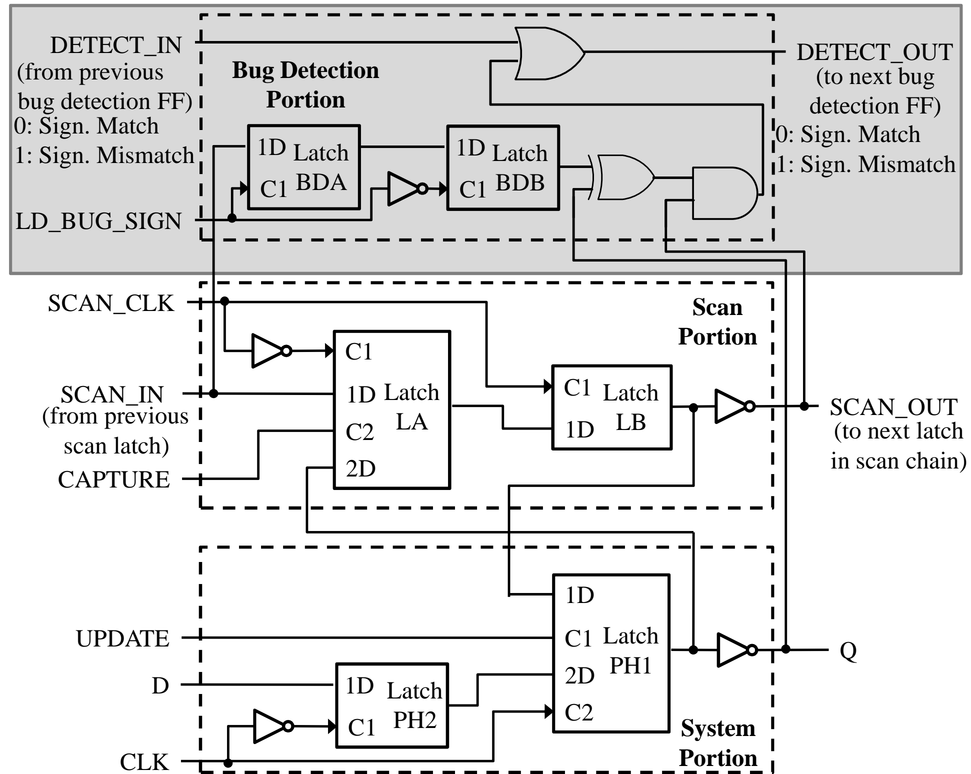


Figure 5.12: Bug Detection Flip-Flop: Modified scan flip-flop with bug detection capabilities.

Hardware Implementation

Bug Detection Flip-Flops: In the ACE-based online design bug detection mechanism, the system bug signature and its comparison with system state is distributed to the flip-flop level. This is achieved by augmenting the system flip-flops with extra logic for storing the system bug signature and comparing it to the system state. Figure 5.12 shows a system flip-flop augmented with these extensions. The non-shaded logic comprises a scan flip-flop, the common type of flip-flop used in modern processors to enable scan-in and scan-out functionality to facilitate manufacturing testing using Automatic Test Pattern Generation [67, 146]. The system portion is used for holding the system state, while the scan portion is used to scan-in test patterns and scan-out test responses. In current designs, the scan portion is utilized only during the manufacturing testing phase and stays idle during normal operation. Also, notice that the non-shaded logic is identical with the ACE flip-flop used in the ACE framework for online defect detection and diagnosis, thus the hardware extension of the ACE flip flop to perform online design bug detection, as illustrated in Figure 5.12, is straightforward.

Specifically, during normal operation, the ACE-based online design bug detection mechanism uses the scan portion in combination with an extra *bug detection portion* to

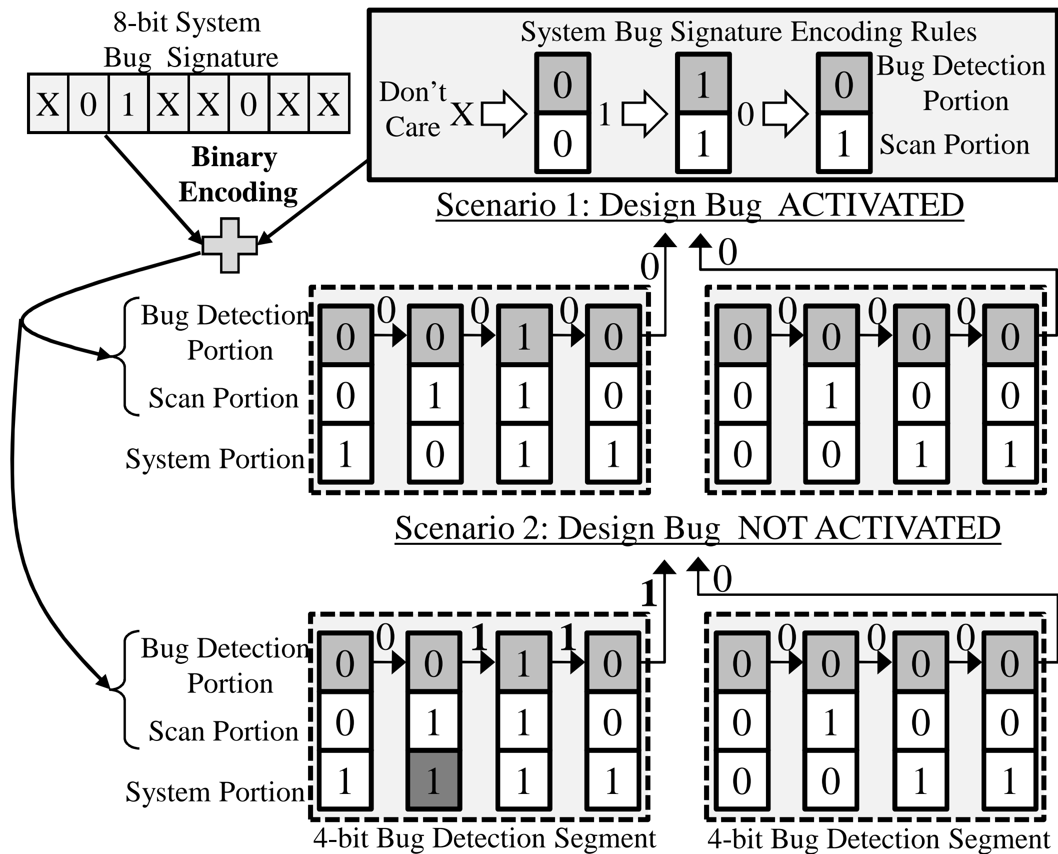


Figure 5.13: Bug Detection Example: Example of an 8-bit bug signature encoding and checking.

store the system bug signature. The scan portion is used to indicate whether the specific flip-flop belongs to any bug's source-level monitor signal set. If the scan portion is set to 1 the flip-flop is indicated as a bug source signal, otherwise the flip-flop's value is irrelevant to the activation of a design bug. In the former case, the value that will activate the design bug is stored in the bug detection portion.

The box at the top of Figure 5.13 illustrates the three encoding rules used to binary map the system bug signature to the bug detection portion (shaded box) and the scan portion (white box). If the scan portion is set, the value of the system flip-flop is compared to the value of the bug detection portion to check if there is a match between the system state and the system bug signature. In our mechanism, flip-flops are grouped into *bug detection segments* to simplify checking; the comparison result is *ORed* with the comparison result of the previous flip-flop to generate a segment-wide match/mismatch signal. The signal is propagated to the next flip-flop (0 indicates a segment match and 1 indicates a segment mismatch). A bug detection segment consists of multiple bug detection flip-flops connected together in a serial fashion (this is analogous to scan segments in scan chains).

Figure 5.13 demonstrates the system bug signature binary encoding process with an example 8-bit system bug signature. The system bug signature is encoded and loaded into the bug detection and scan portions, and the checking is partitioned into two 4-bit bug detection segments. Figure 5.13 also demonstrates how the bug detection segment signals are generated for two different scenarios. In the first scenario, the system state matches with the system bug signature and the segment bug detection signals are both set to `zero` indicating that the bug is activated. In the second scenario, the second bit of the system state does not match with that of the system bug signature and therefore the bug detection signal of the particular segment is set to `one` indicating that the bug is not activated.

Merging Bug Signatures into the System Signature: In this section we describe the technique we employ to merge multiple bug signatures to generate the single system bug signature. First, we merge all the bug signatures related to a single design bug into an intermediate bug signature. To do so, for each bit location we check the values of all bug signatures. If the bit takes the value of `zero` in some signatures and the value of `one` in others, then a `don't care` (X) value is assigned to the merged intermediate bug signature since for that signal either value can lead to a bug triggering combination. If the value of the bit is constant for all signatures then that value is assigned to the merged intermediate bug signature. This technique is illustrated in Figure 5.14 for two example design bugs.

When merging the intermediate bug signatures of multiple bugs into the system bug signature, we employ a slightly different technique. Again, if a bit location takes both values (`one` and `zero`) among different intermediate signatures, it is marked with a `don't care`. The difference from the previous technique is that now it is possible for a bit location to have a `zero` or a `one` in the intermediate signature of one bug and a `don't care` in the intermediate signature of another. This case is treated differently depending on the status of the remaining signals in the bug detection segment:

- *CASE 1:* Consider the two rightmost bits of the middle bug detection segment of Figure 5.14. They both have the value of `one` in one of the intermediate bug signatures and a `don't care` value in the other. Since the whole bug detection segment needs to match to trigger a bug and both bugs have other source signals in this bug detection segment (the second source signal with the value `zero`), the specific source signal is assigned a `don't care` value so that it will not prevent the detection of any of these two particular design bugs.

- *CASE 2:* Now, consider the third bits of the leftmost and the rightmost bug detection segments. Again, in one of the intermediate signatures they have the value of `one`

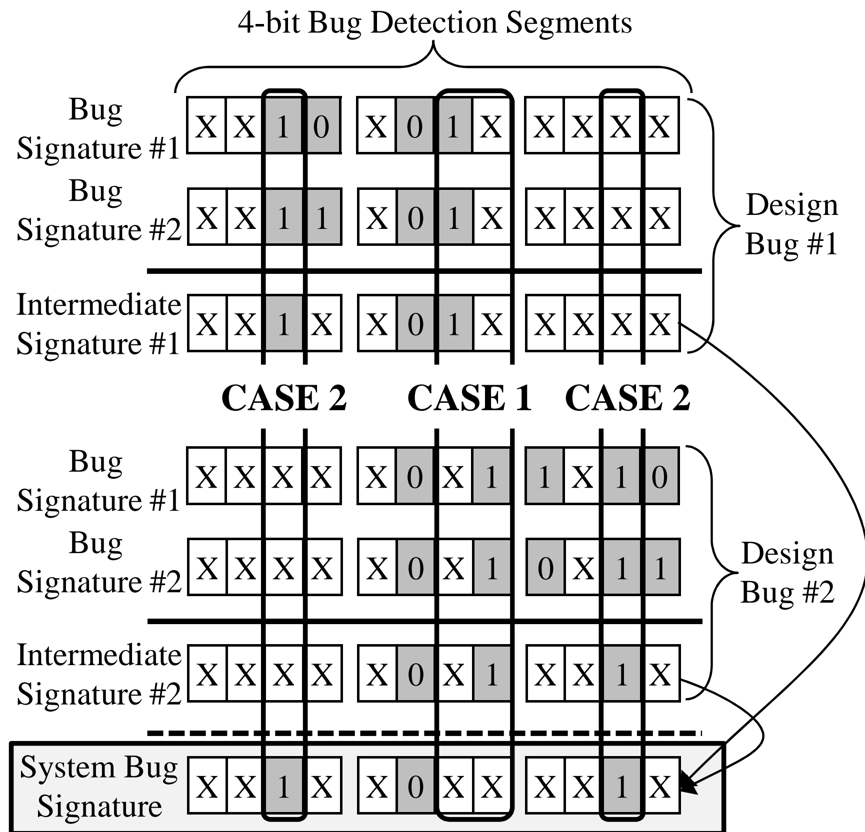


Figure 5.14: Bug Signature Merging: Merging bug signatures into system bug signature

while in the other they have a `don't care` value. However, in this case no other source signal in the bug detection segments is shared between the two bugs. This means that the segments are associated with only one design bug. Therefore, the source signals can be set to `one` in the system bug signature because only a single bug requires the particular segment to match its portion of system bug signature with the system state to detect the bug activation.

False Positives - Notice that our mechanism uses `don't care` values to merge multiple bug triggering conditions and multiple bug signatures. This approach relaxes the bug triggering conditions and can result in *false positives*, that is, non-errant conditions which initiate an innocuous recovery sequence. However, since the technique only relaxes the triggering conditions, it cannot exhibit *false negatives*, that is, discovered design bugs with installed signatures that do not successfully initiate recovery. This is a very important property, since it guarantees that the system will not experience the effects of a specific design bug once the bug is covered by the mechanism.

However, the presence of false positives can adversely impact the performance of the system if too many false recovery alarms are issued. Since the false positive rate highly

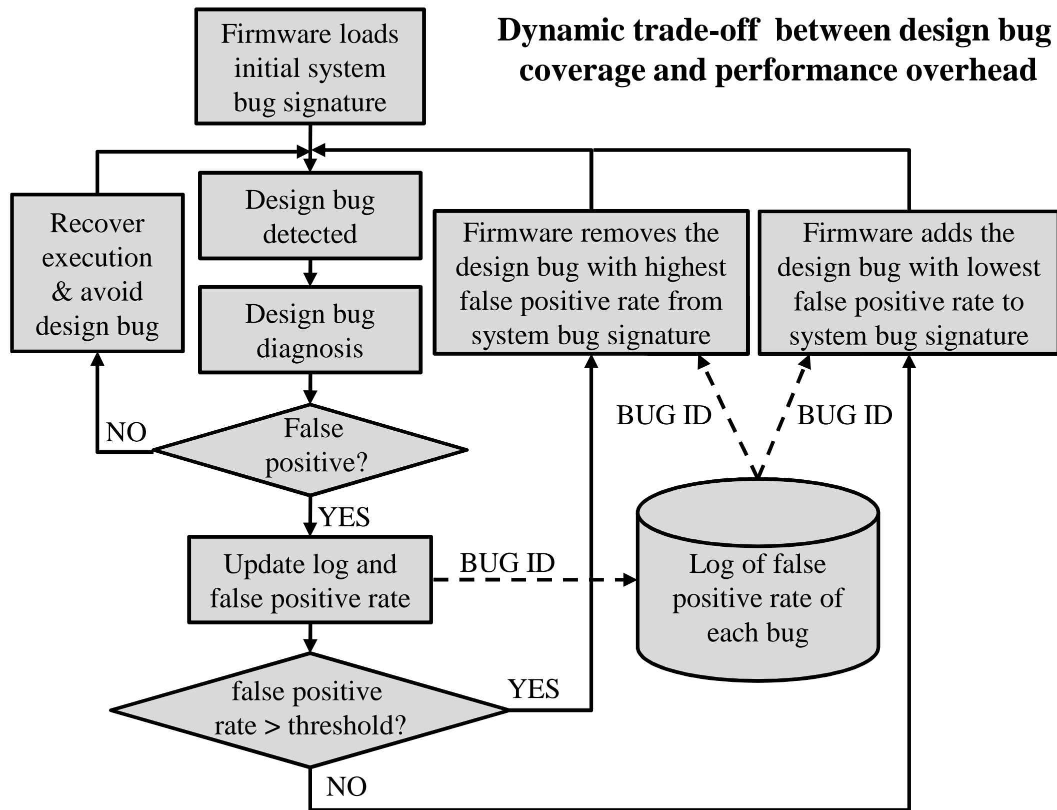


Figure 5.15: Performance/Coverage Trade-off Tuning Algorithm: Dynamically changing the set of covered design bugs to regulate the false positive rate and performance overhead

depends on the dynamic system conditions and workload, we propose a dynamic scheme for trading off design bug coverage with system performance. Figure 5.15 gives a high-level overview of this scheme. At system start-up, firmware loads into the mechanism the initial system bug signature that covers all design bugs. A triggered design bug detection is followed by a diagnosis process that determines if the design bug detection is correct or if it is a false positive. If the detection is correct, the system execution is recovered and the design bug is averted using design bug avoidance techniques. If the detection is a false positive, then the false positive rate of the specific design bug is logged using the bug's ID tag and the system's false positive rate is calculated. The system's false positive rate is then compared with a predefined threshold. If the system's false positive rate is larger than the threshold, the design bug with the highest false positive rate is removed from the set of covered design bugs and firmware regenerates and loads into the mechanism the new system bug signature. On the other hand, if the system's false positive rate is smaller than the threshold, the design bug with the lowest false positive rate is added to the set of covered design bugs.

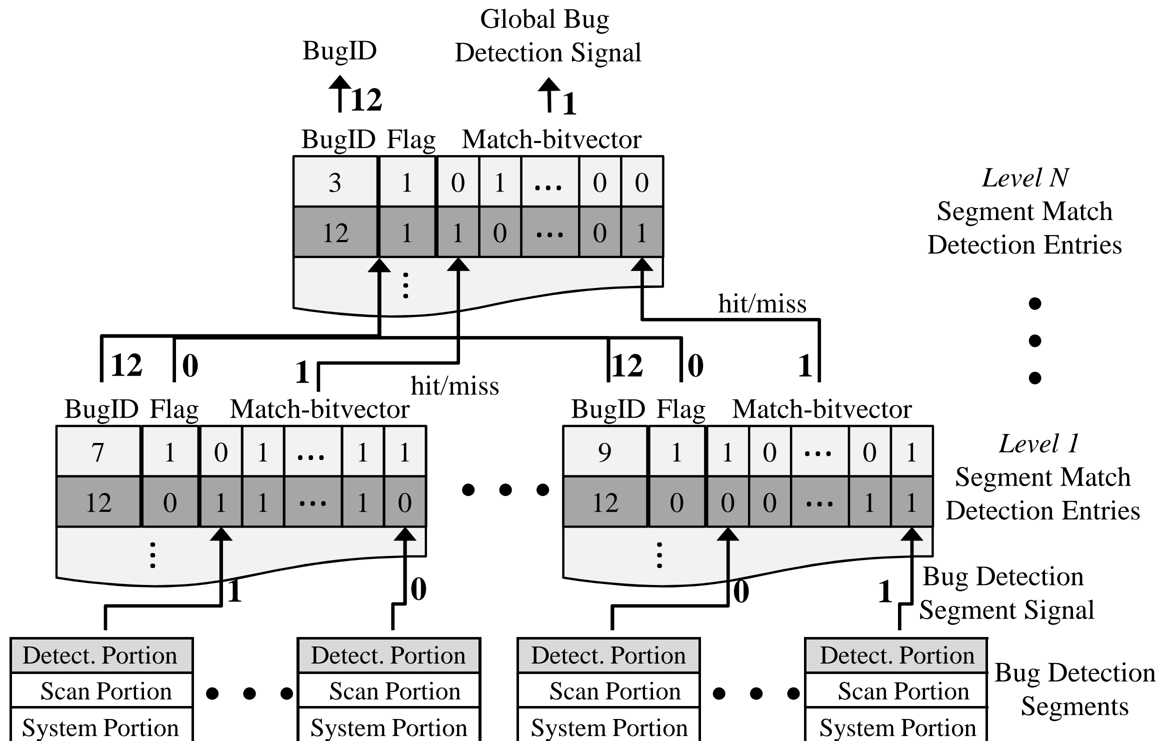


Figure 5.16: ACE-Based Distributed Bug Detection: Example of online design bug detection on an ACE-based distributed segment checking tree.

The predefined threshold can be adapted dynamically based on the requirements of the running applications. For example, a performance-critical application with low dependability requirements can set the threshold low, while a dependability-critical application can set it high. Furthermore, this scheme can be optimized to achieve the optimum trade-off between design bug coverage and performance overhead due to false positives.

5.1.5 ACE-Based Segment Checking Tree Implementation

In the ACE-based online design bug detection mechanism, the bug detection segment signals are aggregated to generate one global bug detection signal through a hierarchical tree structure, the ACE-based segment checking tree. The implementation of this structure is shown in Figure 5.16 and is similar to the implementation of the ACE tree presented in Chapter IV. Each leaf node of the structure is connected to a set of bug detection segments. For each bug that has source signals located in bug detection segments assigned to a leaf node, a *segment match detection entry* is allocated in that node. Each segment match detection entry indicates which subset of the node's bug detection segments need to match the system bug signature to trigger the given bug through the `Match-bitvector` field. Each entry also has a `BugID` and a `Flag` field. The `BugID` field indicates the bug

associated with the specific entry, while the `Flag` field indicates whether the specific bug has source signals that are mapped on a different leaf node.

For example, the design bug with the ID tag 12, has source signals both in the leftmost and in the rightmost leaf nodes of the tree. Therefore, it is allocated a segment match detection entry in each of those nodes with the `Flag` field set to `zero`. On the other hand, the design bugs with ID tags 7 and 9 have source signals limited only to one leaf node and this is indicated by having their `Flag` field set. A bug that has its `Flag` field set means that if the `Match-bitvector` field of that particular bug matches with the values of the underlying bug detection segments, then no further checking is required (since the bug's signals are limited only to that node) and the bug is flagged, along with its ID tag, through the tree to the top level *global bug detection signal*. Notice that if two bugs are flagged in the same cycle (*e.g.*, bugs 7 and 9), only one of them will be flagged to the top level and the decision will be arbitrary based on the implementation. However, due to the rare occurrence of design bugs, we don't expect two design bugs to be triggered in the same cycle.

Figure 5.16 illustrates the detection of the bug with the ID tag 12. In the specific example, the values generated by the underlying bug detection segments match with the `Match-bitvector` fields of bug 12 in both leaf nodes. Since the `Flag` field is set to `zero`, the bug is not flagged and the hit/miss signal from the leaf nodes are passed to the upper level. When the node hit/miss signals reach the top level node of the tree, the values match with the bug's `Match-bitvector` entry and therefore the global bug detection signal is set to `one`, triggering the design bug recovery process, and the bug ID tag is passed to the bug recovery handler.

System-Level Integration

In order to provide a complete online design bug detection solution, the ACE framework presented in the previous chapter offers two additional functionalities:

1. **In-the-Field Programmability:** The system bug signature and the data that need to be stored in segment match detection entries are dynamic and change as new design bugs are discovered or old bugs get fixed. This part of the design needs to be field-programmable and upgradable by special firmware developed and distributed by microprocessor vendors. Since the ACE framework can read/write to any of the tree nodes and any scan flip-flop in the design, it can also be used to program the segment match detection entries in the distributed bug checking tree and load the bug signature at the flip-flop level. Specifically, this functionality is already available in

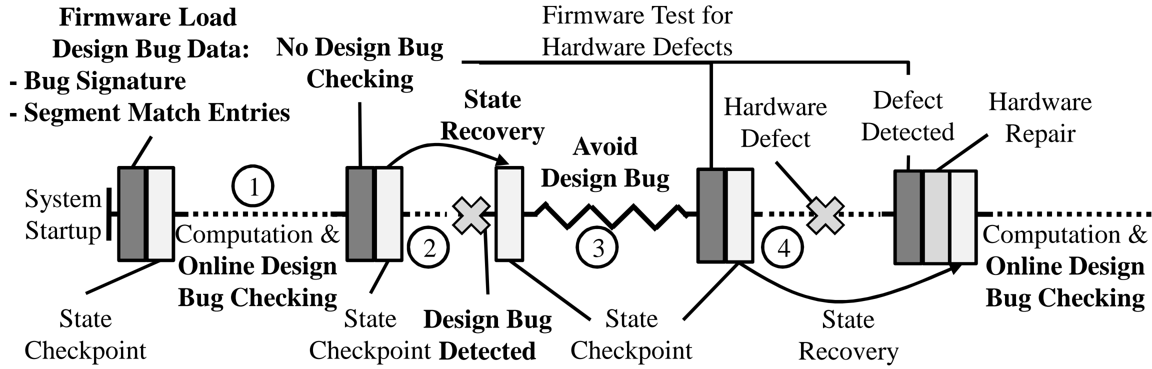


Figure 5.17: ACE Framework for Online Design Bug and Defect Detection: Unifying online design bug detection and silicon defect detection under the ACE framework.

the ACE framework in the form of the ACE instructions, ACE firmware, and ACE tree as described in Chapter IV.

2. **Recovery Support:** The detection of the occurrence of a design bug is only the first step in providing a solution to the problem. Further action is required to avert the design bug and avoid corrupting the execution. This is commonly achieved through recovery support where the system state recovers to the last validated/correct state and execution is guided from there in a way that the design bug is averted. The system state recovery can be provided by the ACE framework since, as described in Chapter IV, it employs coarse-grained checkpoint and recovery techniques to provide system recovery from hardware defects. By rolling back the system state to the last validated and correct system state, execution can be guided by design bug avoidance techniques in a way to avert the design bug. Several design bug avoidance techniques have already been proposed in the research literature [109, 141, 96] and any further advancement toward this direction is not in the scope of this work.

System-Level Operation: The two applications, online design bug detection and online defect detection, can use the ACE framework hardware synergistically and provide a collective solution for reliable and dependable computing. Figure 5.17 shows the synergistic execution timeline of the two applications. At system startup, special firmware uses the ACE framework to load the bug signature and the segment match detection entries needed for online design bug detection. During a checkpoint interval, execution is guarded from the effects of design bugs by the online bug detection mechanism (phase 1). If no design bug is detected, at the end of the checkpoint interval special firmware uses the ACE framework to test the underlying hardware for defects as described in Chapter IV. If the test succeeds, a new checkpoint is taken. If, during the checkpoint interval, a design bug is

detected, the system state is rolled back to the last checkpoint (phase 2) and bug avoidance techniques are employed to avoid the design bug (phase 3). If a hardware defect manifests during a checkpoint (phase 4), the defect is detected at the end of the checkpoint and, after system repair, the system state is rolled back to the last checkpoint for re-execution as described in Chapter IV. Notice that the use of the ACE tree resources and the scan state is mutually exclusive by the two mechanisms. The online design bug detection mechanism utilizes these resources during a checkpoint interval, while the hardware defect detection mechanism utilizes the resources at the end of a checkpoint interval. Hence, the cost of the ACE framework is amortized between bug detection and defect detection.

5.1.6 Experimental Evaluation

Experimental Methodology

The case study design used for the experimental evaluation of our mechanism is the OpenSPARC T1 chip-multiprocessor, the open-source version of Sun's Niagara (UltraSPARC T1) [126]. We choose this design because the OpenSPARC T1 chip-multiprocessor targets commercial applications such as database and web servers where system correctness is of paramount importance. We believe such systems constitute ideal candidates to employ our mechanism to provide the required correctness guarantees. The OpenSPARC T1 is a full-system multiprocessor design implementing the 64-bit SPARC V9 architecture. It contains eight 6-stage pipelined in-order cores, each with 8KB L1 data-cache, 16KB L1 instruction-cache and full hardware support for four threads. The eight cores are connected through a crossbar to a unified 3MB L2 cache and a shared floating-point unit. The chip also includes four memory controllers and an input/output bridge [127].

RTL Implementation: To make an accurate assessment of our mechanism's requirements in silicon area and power consumption, we developed a detailed RTL model of our mechanism in Verilog. Specifically, in our prototype we implemented 1) the bug detection flip-flops that hold the bug signature and compare it with the system state, 2) the segment checking tree with a parameterized number of segment match detection entries per tree node, and 3) the ACE-based field programmable framework that loads through firmware the bug signature and the segment match detection entries. Our implementation covers all modules in OpenSPARC T1 except the memory cache data and tag arrays (we don't expect logic design bugs to be located in regular and meticulously optimized arrays).

Logic Synthesis and Tools: We used the Synopsys Design Compiler to perform logic synthesis on the RTL code of the OpenSPARC T1 and our mechanism. Logic synthesis mapping is done using the Artisan IBM 0.13um standard cell library. The resulting gate-

Methodology/Tools Used	Design Components
Synopsys Power Compiler	1) SPARC Cores, 2) Crossbar, 3) FPU, 4) Misc. Units (I/O Bridge, DRAM Controllers, Control & Test Unit) 5) ACE Framework, 6) Online Design Bug Detection Mechanism
CACTI 4.2	1) L1 Inst. & Data Caches, 2) L2 Cache
Taken from [72]	1) I/O Pads, 2) Wires & Repeaters

Table 5.2: Power Consumption Estimation Methodology: The table lists the methodology/tools used to estimate the power consumption of the major OpenSPARC T1 components.

level netlists of the OpenSPARC design and our mechanism provided a common substrate to accurately estimate the silicon area and power consumption overhead on the whole OpenSPARC design.

Power Consumption Estimation Methodology: To evaluate the power consumption overhead of our mechanism, we first estimated the power consumption of the baseline OpenSPARC T1 design without the extra hardware required by our mechanism. We calibrated the estimated power consumption with actual power consumption numbers provided by Sun for each module of the chip [72]. After we validated our power estimates for the baseline OpenSPARC T1 design, we estimated the additional power required by our mechanism. Table 5.2 shows the major design components of the OpenSPARC T1 and the methodology/tools we used to estimate their power consumption. We estimated the power consumption of the majority of the OpenSPARC T1 modules using the Synopsys Power Compiler (part of the Synopsys Design Compiler package). To estimate the power consumption of the L1 and L2 caches, we used the CACTI 4.2 tool [132], a tool with integrated performance, area, and power models for memory cache structures.

This methodology is sufficient to estimate the power consumption of most of the chip's logic modules. However, there are parts of the design whose power consumption cannot be accurately estimated with these tools. These include 1) numerous buses, wires, and repeaters distributed all over the design, which are very hard to model accurately using the Power Compiler, unless the design is fully placed and routed, 2) I/O pads of the chip. In order to estimate the power consumption of these two parts, we used values from the reported power envelope of the commercial Sun UltraSPARC T1 design [72].

Area Overhead and Design Bug Coverage

Control vs. Data Signals - After synthesizing the OpenSPARC T1 chip we found that there are about 262K flip-flops in the design. We also found that providing monitoring and

Chip Submodule	Data Signals	Control Signals
SPARC Core (x8)	15632 (79.06%)	4140 (20.94%)
CPU-Cache Crossbar	27283 (98.69%)	362 (1.31%)
Floating-Point Unit	4054 (87.75%)	566 (12.25%)
Control & Test Unit	2325 (55.29%)	1880 (44.71%)
Input/Output Bridge	10251 (95.14%)	524 (4.86%)
DRAM Controller (x4)	13449 (94.70%)	752 (5.30%)
Total	222765 (84.95%)	39460 (15.05%)

Table 5.3: Data and Control Signals in OpenSPARC T1: The table shows the percentage of data and control signals in the OpenSPARC T1 processor.

bug detection capabilities for all these signals results in prohibitive area overhead (~69%). However, we observed that the majority of these flip-flops serve as buffers to data buses or data registers, and only a small fraction of them are control signals. Furthermore, after analyzing the source signals of the logic design bugs studied in Section 5.1.3, we found that all of the bug source signals were control signals, and no logic design bug had a source signal that was part of a data bus or a data register. After this observation, we partitioned the flip-flops of the OpenSPARC T1 design into *data* and *control* signals. Table 5.3 shows the fraction of data and control signals for all modules in the OpenSPARC T1. For the whole chip, only 39K flip-flops drive control signals, accounting for 15% of all flip-flops in the design.

Our prototype implementation taps all 39K control signals in the OpenSPARC T1 design. This means that each of these flip-flops is augmented with the extra bug detection logic shown in Figure 5.12. The area overhead of this flip-flop augmentation is estimated to be 3%. Flip-flops are grouped into 8-bit bug detection segments and connected to a four-level segment checking tree structure (shown in Figure 5.16). The area overhead of the tree structure depends on the number of segment match detection entries per tree node. The number of design bugs that can be covered by our mechanism also depends on the number of segment match detection entries per tree node, raising an engineering trade-off between area overhead and bug coverage.

Area Overhead vs. Coverage - Figure 5.18 illustrates this trade-off based on the 162 logic design bugs located in the SPARC core's LSU and TLU units studied in Section 5.1.3. The figure depicts the percentage of design bugs covered (left Y-axis) and the area overhead (right Y-axis) versus the number of segment match detection entries per tree node. When the tree nodes are equipped with 32 entries, our mechanism can cover all the 162 design bugs with an overall area overhead of 17%. Fortunately, not all design bugs

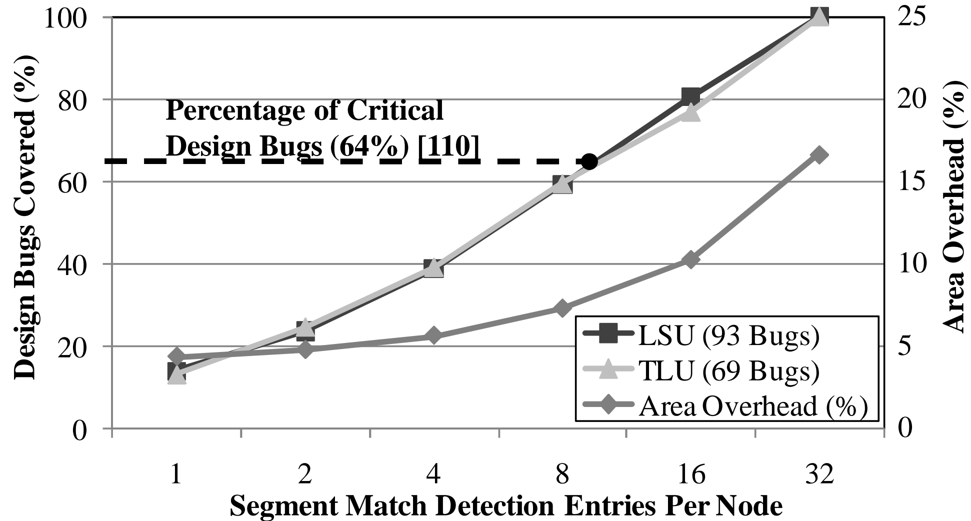


Figure 5.18: Area Overhead Versus Design Bug Coverage: The graph illustrates the trade-off between area overhead and design bug coverage. As the number of segment much detection entries per tree node is increasing, so does the design bug coverage and the area overhead.

are critical to functional correctness and need to be covered. Sarangi et al. [110] studied the errata documentation of ten modern microprocessors and found that, on average for all the studied processors, 64% of the design bugs are critical to functional correctness. The remaining 36% of the design bugs were found to be non-critical to the correctness of the system and commonly located in modules such as performance counters, error reporting registers, or breakpoint support [110]. In Figure 5.18, we can observe that 16 segment match detection entries per tree node provide a design bug coverage of 80% that is much higher than the typical fraction of critical design bugs. This design configuration leads to a silicon area overhead of 10% of the whole OpenSPARC T1 design.

Power Consumption Overhead

Employing the methodology described in Section 6.5.1, we estimated the power envelope of the baseline OpenSPARC T1 chip, without the additional hardware required by our mechanism, to be 56.3W. Our estimate of the OpenSPARC T1 power is within 12% of the reported power consumption of the commercial Sun Niagara design [72]. Figure 5.19 shows the power consumption for our enhanced OpenSPARC T1 design including our online design bug detection mechanism. The power envelope of the enhanced design is 58.3W. From this, a total of 3.4% (1.96W) is devoted to the extra hardware required by our mechanism. The overall power consumption overhead of our mechanism over the baseline is therefore about 3.5%.

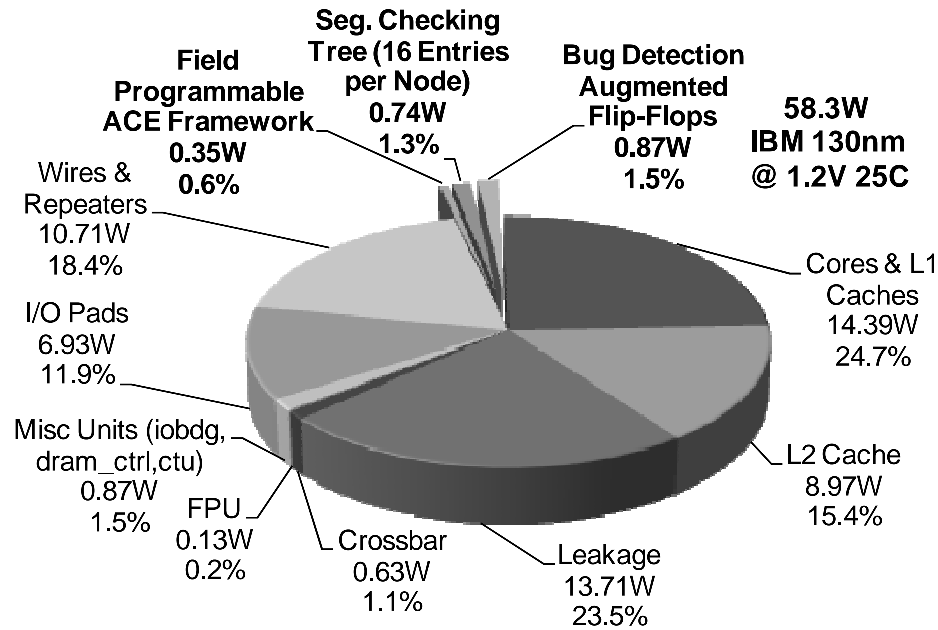


Figure 5.19: Power Consumption Overhead: The pie chart shows the power consumption of the OpenSPARC T1 processor augmented with the ACE-based online design bug detection mechanism.

Mechanism	Flip-Flops Covered	Area Overhead	Power Overhead
Online Design Bug Detection (16 seg. comparator entries per tree node)	39K Flip-Flops	10.26%	3.5%
Online Hardware Defect Detection	262K Flip-Flops	5.8%	4%
Online Design Bug Detection + Online Hardware Defect Detection	39K Flip-Flops (bug detection) 262K Flip-Flops (defect detection)	15.15%	6.8%

Table 5.4: Overhead of the Extended ACE Framework: The table shows the total overhead of the combined design bug and defect detection ACE framework.

Unified Design Bug & Defect Detection Overhead

Table 5.4 presents the silicon area and power consumption overhead of the extended ACE framework. The estimated silicon area overhead of the extended framework is 15.15%, and its power consumption overhead is 6.8%. Based on these numbers, we believe that the extension of the ACE framework hardware to provide online design bug detection, in addition to Online defect detection and diagnosis, is an attractive and relatively low overhead solution for high-dependability computing.

5.2 Related Work

Design Bug Analyses: Our online design bug detection mechanism is based on insights from this work and previous design bug analyses that characterize the known design bugs of existing processors. Section 5.1.2 provides a discussion on previous design bug analyses and how our RTL design bug analysis differs from those previous works.

Online Design Bug Detection: Recently, several works proposed the use of online, in-the-field design bug detection and avoidance as a mechanism to mitigate the negative effects of design bugs [110, 96, 141].

Specifically, Sarangi *et al.* [110] propose the Phoenix, a field-programmable mechanism that continuously taps key logic signals to detect the occurrence of design bugs while the processor is operating in the field. In particular, Phoenix uses a software structure at the supervisor level, called the signature buffer, to hold the triggering conditions of design bugs. The supervisor uses the triggering conditions stored in the signature buffer to program the field-programmable portion of the Phoenix mechanism. The field-programmable portion of Phoenix consists of two components: i) the Signal Selection Unit (SSU), a switch that is made out of programmable pass transistor and selects the logic signals that need to be monitored for the detection of the occurrence of a specific design bug, and ii) the Bug Detection Unit (BDU), a logic array that combines signals selected by the SSU into logic expressions that flag the occurrence of a design bug. The input signals to the SSU are a set of control signals selected at designs time, that at the designers' judgment could possibly help the detection of any yet unknown design bugs.

The Phoenix mechanism is partitioned into several subsystems, each with a local SSU and BDU, that are distributed to the different microarchitectural components. Each Phoenix subsystem uses a hub to collect monitored signals from the local SSU and pass them over to the hubs of other neighboring subsystems, and to bring in signals from other hubs to the local BDU.

Wagner *et al.* in [141] proposed FRCL, a field-programmable mechanism for the online detection and recovery of design bugs. In FRCL, the online detection of design bugs is performed by routing a set of signals that are selected at design time to a centralized state matcher, a fully-associative field-programmable array that holds the bit-patterns that represent the triggering conditions of design bugs. In order to limit the number of the state matcher entries, the state matcher is structured in way to allow the use of "don't care" values in the bit-patterns. This extension, enables optimizations like the combination of similar design bug bit-patterns into a single bit-pattern. These optimizations were used for the development of a pattern-compression algorithm with the goal of reducing the number

of entries required for the state matcher. In the same work, Wagner *et al.* also developed a software tool for the automatic selection of the set of signals to be monitored. Specifically, in order to find the more critical set of signals to monitor, the tool considers the RTL description of the design and it ranks signals based on the width of the cone of logic that a signal drives and the number of submodules that they feed into. The proposed tool helps the designers to choose the set of critical signals to be monitored for the online detection of design bugs and can potentially improve the effectiveness of FRCL by predicting which design signals would be involved in the triggering conditions of future, yet undiscovered, design bugs.

A field-programmable approach for online design bug detection was also proposed by Narayanasamy *et al.* in [96]. The approach proposed in [96] differs from the other online design bug detection techniques and the technique proposed in this thesis because of its capability of detecting design bugs with triggering conditions that span across multiple clock cycles. Specifically, in [96], the triggering conditions of each design bug are represented as a combination of set of events that happen in a specific time interval, where events are signals with a particular value. As design bug triggering events occur, they are reported to a monitoring unit that is programmed with all the combinations of events and the time interval that these events need to occur for each particular design bug to be triggered. Each event is reported to the monitoring unit with a timestamp and if the monitoring unit determines that all the triggering events of a particular design bugs occurred in the specified time interval, the occurrence of the design bug is effectively detected and recovery is initiated.

Contribution Over Previous Work: In all these previously proposed mechanisms, the online design bug detection is facilitated by a signal monitoring substrate. However, in all these works the signal monitoring substrate is limited to a small set of signals selected at design time when the design bugs are still unknown. With this approach, if a design bug is discovered after the final release of a microprocessor and its bug triggering conditions involve signals not included in the original set of signals that was selected to be monitored by the substrate, the occurrence of the design bug cannot be detected. In some cases, such design bugs can be detected by over-approximating the bug triggering conditions using the the original set of signals that was selected to be monitored, but this can lead to a high rate of false positives and high runtime performance overhead due to the false recoveries. This means that the effectiveness of these online design bug detection mechanisms depends on decisions made at design time based on assumptions regarding the set of signals that would be involved in yet unknown design bug triggering conditions. This constitutes a major limitation for the effectiveness of these previously proposed mechanisms. The ACE

framework hardware extensions used for the online detection of design bugs address this limitation and improve on these previous works with a novel field-programmable substrate that is capable of monitoring *all control signals in the design* that can trigger a design bug. This capability waives the requirement of selecting the set of signals to be monitored at design time and allows this decision to be made after the product release and the discovery of design bugs, when the design bugs and their triggering conditions are known.

Bug Avoidance Techniques: After the occurrence of a design bug has been detected, the next action that needs to be taken is to avoid any effects on correct execution. Since the design bug is detected a few cycles after its occurrence, the system state first needs to be rolled back to the last correct state before the design bug occurrence, and then execution has to be repeated. The goal during this second execution iteration is to employ techniques that avoid exercising the "buggy" part of the design and avert another occurrence of the design bug. In the research literature, there are already several design bug avoidance techniques proposed [109, 141, 96, 110]. Below we provide a brief description of some of these techniques:

- *Degradation to a formally-verified mode*: Wagner *et al.* in [141], proposed that once the execution has been rolled back and the system state has been recovered, execution switches to a simpler (lower-performance), formally verified safe-mode that is free of design bugs. The execution is resumed to the normal mode of operation once it passes the point where the design bug occurrence was detected.

- *Replay after pipeline flush*: If the design bug can be detected before its effects corrupt the architectural state, then a pipeline flush might be adequate to change the order of execution events that triggered the design bug [109]. Several techniques have been proposed on how to change the order of execution events, such as adding extra NOPs between instructions [109].

- *Replay after checkpoint recovery*: Conceptually, this technique is the same as the replay after pipeline flush technique, but this technique also recovers the architectural state in case that the design bug was detected late and its effects could have corrupted the architectural state [109].

- *Instruction-stream editing*: This technique overrides the BIOS microcode of specific instructions with a new sequence of micro-operations that avoid exercising the "buggy" part of the design [109, 141].

- *Hypervisor-guided execution*: In this technique, after the system state has been recovered, execution traps to the hypervisor. In many cases, the hypervisor is capable

of intercepting and interrupting the control flow of execution and taking sophisticated corrective actions in a way to guide execution past the the point where the design bug was detected without triggering the detected design bug [109].

Dynamic Validation: Another approach to deal with the design bugs found in modern microprocessors is the dynamic validation of the execution while the microprocessor is operating in the field by adding some extra on-chip checkers. In particular, these on-chip checkers continuously monitor the microprocessor execution and check if the execution steps into a non-validated state, or detect execution errors that were caused by design bugs. An example of a dynamic validation checker is DIVA, an online checker component, in the form of a very simple core, that is inserted into the retirement stage of a microprocessor pipeline that continuously validates the computation, communication, and control exercised in a complex out-of-order microprocessor core [6, 143].

In this context, more recently, Wagner *et al.* proposed the concept of semantic guardians in [139] to guarantee bug-free and functional correct execution in microprocessor designs. A semantic guardian is a hardware component that is automatically synthesized based on the microprocessor's functional validation coverage data and it is included in the microprocessor design. At runtime, the guardian monitors a subset of the design's internal signals. If the guardian detects that the system steps in a non-validated configuration, it switches execution into a lower-performance but formally verified safe-mode version of the microprocessor to guarantee functionally correct execution.

Another on-chip checker is Chico, presented by DeOrio *et al.* in [30]. Chico focuses on the dynamic validation of control logic by monitoring the flow of instructions executed by the processor. Specifically, Chico is targeting to detect execution errors that manifest in the control aspects of the execution like data forwarding and branching selection. Similar to the semantic guardians approach, when Chico detects an execution error, it switches execution into a formally verified, lower-performance execution mode until the offending instruction that caused the error is committed.

Other dynamic validation solutions proposed in the research literature include the work by Meixner *et al.* [86] that detects execution errors caused by design bugs in the dataflow circuitry by dynamically verifying high-level invariants that error-free executions are guaranteed to maintain, and the work by Chen *et al.* [25] that uses constraint graph models to dynamically validate the end-to-end correctness of a transactional memory system.

One of the drawbacks of dynamic validation when compared to online design bug detection is that specific on-chip checkers need to be designed for the validation of each functional task of the microprocessor, which results into a higher complexity and more intrusive solution to address design bugs. In contrast, online design bug detection can

provide a comprehensive solution that addresses the design bugs of a microprocessor design as a whole. Another limitation of dynamic validation techniques like the semantic guardians [139] is that they treat all non-validated system configurations as potential design bugs and trigger system recovery and a switch into a lower-performance safe-mode execution. In a complex system with a lot of non-validated system configurations, this can result to significant performance overhead. On the other hand, dynamic validation solutions can provide functional correctness against design bugs that are not yet discovered.

5.3 Other Applications of the ACE Framework

We believe that the ACE framework is a general framework that can be extended to several other applications to amortize its hardware cost. Specifically, its capability to provide hardware accessibility and controllability to the software can find use in many applications. In this section, we describe how the ACE framework can be extended to improve two important phases of the microprocessor design cycle. Specifically, Section 5.3.1 describes how the ACE framework hardware can be extended and used as a tool to ease the post-silicon debugging process, while Section 5.3.2 describes how the ACE framework can improve the microprocessor manufacturing testing.

Notice that today, for none of these two applications the area overhead of the ACE framework would be justifiable. However, if the area overhead of the ACE framework can be justified by the need to provide defect tolerance to the microprocessor design (as it was proposed in Chapter IV), and for the additional capability of online design bug detection (as it was proposed in Section 5.1), then the extension of the ACE framework to these applications comes for free as an additional feature and adds value to the ACE framework. This additional value and extra capabilities can ease the potential adoption of the ACE framework in future generation microprocessors, as it would be possible to use the framework's hardware resources to address multiple problems.

5.3.1 ACE Framework Extensions for Post-silicon Debugging

Post-silicon debugging is an essential and highly resource-demanding phase that is on the critical path of the microprocessor development cycle. Following product tape-out (*i.e.*, the fabrication of the microprocessor into a silicon die), the post-silicon debugging phase checks if the physical design of the product meets all the performance and functionality specifications as they were defined in the design phase. The goal of post-silicon debugging is to find all design errors, also known as design bugs, and to eliminate them through design changes or other means before selling the product to the customer [60, 49, 61].

The first phase of post-silicon debugging is to run extended tests to validate the functional and electrical operation of the design. The validation content commonly consists of focused software test programs written to exercise specific functionalities of the design or randomly generated tests that exercise different parts of the design. We refer to these test programs as the *validation test suite*. These tests are applied under different operating conditions (*i.e.*, voltage, clock frequency, and temperature) in order to electrically characterize the product. When the observed behavior diverges from the expected pre-specified correct behavior (*i.e.*, when a failure is found), further investigation is required by the post-silicon debugging team. During a failure investigation the post-silicon debug engineer tries to i) isolate the failure, ii) find the root cause of the failure, and iii) fix the failure, using features hardwired into the design to support debugging as well as tools external to the design [60].

Motivation: The trends of higher device integration into a single chip and the high complexity of modern processor designs make the post-silicon debugging phase a significantly costly process, both in terms of resources and time. For modern processors, the post-silicon debugging phase can easily cost $15 to $20 million and take six months to complete [39]. The post-silicon debugging phase is estimated to take up to 35% of the chip design cycle [24], resulting in a lengthy time-to-market. As the level of device integration continues to rise and the complexity of modern processor designs increases [35], this problem will be exacerbated leading to either i) very expensive and long post-silicon debugging phases, which would adversely affect processor's cost and/or time-to-market or ii) more buggy designs being released to the customers due to poor post-silicon debugging [140, 109], which would likely increase the fraction of chips that fail in the field.

There are two major challenges in the post-silicon debugging process of modern highly-integrated processors. First, because the internal signals of the microarchitecture have limited observability to the testing software, it is difficult to isolate a failure and find its root cause. Second, because the hardware design is not easily or flexibly alterable by the post-silicon debug engineer, it is difficult to evaluate whether or not a potential fix to the design eliminates the cause of the failure [61]. Existing techniques that are used to address these two challenges are not adequate, as briefly explained below.

Traditional techniques used to address the limited signal observability problem are built-in scan chains [146, 61] and optical probing tools [149]. Unfortunately, both have significant shortcomings. The use of built-in scan chains to monitor internal signals is very slow due to the serial nature of external scan testing [45], which is part of the reason why post-silicon debugging takes a significant fraction of the processor design cycle. The effectiveness of optical probing tools reduces with each technology generation as direct probing becomes very difficult, if not impossible, with more metal layers and smaller

devices [137]. Furthermore, it is very hard to integrate these two techniques into an automated post-silicon debugging environment [137].

The traditional technique used to evaluate design fixes is the Focused Ion Beam (FIB) technique [60], which temporarily alters the design by physically changing the metal layers of the chip. Unfortunately, FIB is limited in two ways. First, FIB typically can only change metal layers of the chip and cannot create any new transistors. Therefore, some potential design fixes are not possible to make or evaluate using this technology. Second, FIB's effectiveness is projected to diminish with further technology scaling as the access to lower metal layers is becoming increasingly difficult due to the introduction of more metal layers in modern designs [24, 60].

Recently proposed mechanisms try to address the limitations of these traditional techniques. Specifically, recently proposed solutions suggest the use of reconfigurable programmable logic cores and flexible on-chip networks that will improve both signal observability and the ability to temporally alter the design [102]. However, these solutions have considerable area overheads [102] and still do not provide complete accessibility to all of the processor's internal state [102].

Solution - ACE Framework Extensions for Post-silicon Debugging: The ACE framework can be an effective low-overhead framework that provides the post-silicon debug engineers with full accessibility and controllability of the processor's internal microarchitectural state at runtime. This capability can be helpful to post-silicon debug engineers in isolating design bugs and finding their root causes. Furthermore, once a design bug is isolated and its causes have been identified, the ACE framework can be used to dynamically overwrite the microarchitectural state and thus emulate a potential hardware fix. This allows the debug engineer to quickly observe the effects of a potential design fix and verify its correctness without any physical hardware modification.

Specifically, the event that triggers a failure investigation by a post-silicon debug engineer is an incorrect design output during the execution of the validation test suite. However, by just observing the incorrect output it is very hard to pinpoint the root cause of the failure.[4] Therefore, further debugging of the failure is required. The first step in this process is the reproduction of the conditions under which failure occurred. Once the failure is reproduced, debugging tools can be used to analyze the design's internal state and pinpoint the design bug. This is where the ACE firmware could be very useful to a post-silicon debug engineer. The debug engineer can run the ACE firmware as an independent thread (called the ACE debugging thread) that runs in conjunction with the validation test thread to identify the root cause of the failure and evaluate a potential design fix. We first

[4] As is also the case for buggy software.

Post-silicon Debugging ACE Instructions
ACE_pause <# of clock cycles> Pauses the execution of the running validation test thread after it is executed for a given number of clock cycles, and switches execution into the ACE debugging thread.
ACE_return Returns execution from the ACE debugging thread to the validation test thread and swaps the scan state with the processor state in order to restore the microarchitectural state of the validation test thread.

Table 5.5: ACE Instruction Extensions for Post-Silicon Debugging: Additional ACE instruction set extensions for post-silicon debugging.

describe the required extensions to the ACE framework to support post-silicon debugging using the ACE firmware, then provide a detailed example of how the debug engineer uses the ACE framework.

ACE Instructions for Post-Silicon Debugging: Table 5.5 shows the ACE instruction set extensions that enable the synchronization between the validation test thread and the ACE debugging thread.

The `ACE_pause` instruction pauses the execution of the running validation test thread after it is executed for a given number of clock cycles, and switches execution to the ACE debugging thread. The execution switch between the validation test thread and the ACE debugging thread is scheduled by setting an interrupt counter to the parameter value of the `ACE_pause` instruction. This interrupt counter decrements every clock cycle during the execution of the validation test thread. Once the counter becomes zero, the processor state and scan state get swapped, thus taking a snapshot of the running microarchitectural state of the validation testing thread into the scan state. In the same clock cycle, execution is switched to the ACE debugging thread.

The `ACE_return` instruction returns execution from the ACE debugging thread to the validation testing thread and swaps the scan state with the processor state in order to restore the microarchitectural state of the validation test thread.

Post-Silicon Debugging Example using the ACE Framework: Figure 5.20 shows an example of a possible ACE firmware written to perform post-silicon debugging. The example firmware is written by the post-silicon debug engineer. Suppose that the debug engineer runs a validation test program that fails after ten thousand cycles of execution, and the validation engineer suspects that the bug is in the third ACE domain of the core. Figure 5.20 shows the pseudo-code of the ACE firmware written to analyze such a failure. The first portion of the code (Figure 5.20-top left) pauses the execution of the validation test program at the desired clock cycle; the second portion (Figure 5.20-top right) allows the debug engineer to single-step the execution by one cycle to observe state changes.

Based on the information obtained by running these portions of the code, the engineer devises a possible fix. The third portion of the code (Figure 5.20-bottom center) is used by the engineer to evaluate whether or not the design fix would result in correct execution. We describe each code portion of the ACE firmware in detail below.

The debugging process starts with the execution of the ACE debugging firmware thread (Figure 5.20-top left). In this thread, the first instruction is an `ACE_pause` instruction that sets the interrupt counter to the clock cycle in which detailed debugging is desired by the post-silicon debug engineer. In the example shown in Figure 5.20, the validation test is set to be interrupted at clock cycle ten thousand (assuming that this is the phase of the validation test where the post-silicon debug engineer suspects that the first error occurs). The `ACE_pause` instruction is followed by an `ACE_return` instruction. `ACE_return` switches execution from the ACE debugging thread to the validation test thread and thus the validation test program's execution begins.

After ten thousand cycles into the execution of the validation test thread, the validation test thread is interrupted. At this point, 1) processor state is swapped with the scan state, and 2) execution is switched from the validation test thread to the ACE debugging thread. Once execution is transferred to the ACE debugging thread, the post-silicon engineer uses the ACE framework to investigate the microarchitectural state of the validation test thread during clock cycle ten thousand (which is stored in the scan state). The example scenario in Figure 5.20 assumes that the suspected bug is in the third ACE domain of the core. `ACE_get` instruction reads the third ACE domain's microarchitectural state and prints it to the debugging console. We assume that the domain's microarchitectural state is checked by the debug engineer and is found to be error-free. Therefore, the debug engineer decides to check the domain's state in the next clock cycle. In order to step the execution of the validation test thread for one clock cycle, the interrupt counter is set to one using the `ACE_pause` instruction, and the validation test thread's execution is resumed with the execution of the `ACE_return` instruction (Figure 5.20-top right).

After one clock cycle of validation test execution, control is transferred again to the ACE debugging thread and the domain's new microarchitectural state is checked by the debug engineer. After inspecting the domain's microarchitectural state, the debug engineer finds that the third bit of the domain's sixth segment is a control signal that should be a zero but instead it has the value of one. Thus, the engineer pinpoints the root cause of the failure. In order to verify that this is the only design bug that affects the execution of the validation test thread, and that fixing the specific control signal does not cause any other erroneous side effects, the debug engineer modifies the domain's microarchitectural state and sets the control signal to its correct value using the `ACE_set` instruction (Figure 5.20-

1.Pause Execution Read Processor State	**2. Step for one cycle Read Processor State**

ACE_pause 10000
ACE_return
// Continue running the validation
// test for 10000 cycles
...
// 10000 cycles later the processor
// state and scan state are swapped
// and the ACE thread is resumed
 	
// The bug is suspected to be in
// domain#3. Read and print
// domain's state for cycle 10000
for(j=0;j<#_of_ACE_Segments;j++){
  ACE_get $r1, 3, j
  print $r1
}
	// Set the interrupt counter
	// to step for one cycle
	ACE_pause 1
	// Swap processor state with
	// scan state and resume the
	// validation test execution
	ACE_return
	
	// After one cycle of validation
	// test execution the ACE
	// debugging thread is resumed
	
	// Read and print domain's
	// state for cycle 10001
	for(j=0;j<#_of_ACE_Segments;j++)
	{
	  ACE_get $r1, 3, j
	  print $r1
	}

3. Fix Buggy State
Continue Execution

```
// Bug found by debug engineer at the state of cycle 10001.
// A control signal should be 0 instead of 1 in segment#6 bit 3.
// Modify processor state to check if bug is fixed.
ACE_get  $r1,3,6
and      $r1,$r1,FFFFFFF7
ACE_set  $r1,3,6

// Run the rest of the validation test
ACE_pause 90000
// Swap processor state with scan state and resume execution
ACE_return
...
// At the end of validation test check if bug is fixed
```

Figure 5.20: ACE Firmware for Post-Silicon Debugging: Example ACE firmware pseudo-code used for post-silicon debugging.

bottom center). Assuming that the whole validation test takes one hundred thousand clock cycles to execute, the debug engineer sets the next debugging interrupt to occur after ninety thousand clock cycles, which is right after the completion of the validation test. At this point, the execution is transferred to the validation test thread, which runs uninterrupted to completion. After completion, the debug engineer checks the final output to verify that the potential design bug fix led to the correct output and there were not any erroneous side effects due to the introduction of the bug fix. In the case that the final output is incorrect, a new failure investigation starts from the beginning and the debug engineer writes another piece of firmware to investigate the failure.

We would like to note the analogy between ACE framework based post-silicon debugging and conventional software debugging. `ACE_pause` instruction is analogous to setting a breakpoint in software debugging. `ACE_return` is analogous to the low-level mechanism that allows switching from the debugger to the main program code. Examining the state of the processor and stepping hardware execution for one cycle are analogous to examining the state of program variables and single stepping in software debugging. Finally, ACE framework's ability to modify the state of the processor while the test program is running is analogous to a software debugger's ability to modify memory state during the execution of a software program that is debugged. We note that, similarly to a software debugging program, a graphical interface can be designed to encapsulate the post-silicon debugging commands to ease the use of ACE firmware for post-silicon debugging.

Advantages: The results of this detailed debugging process, demonstrated by the above example, are sometimes achievable using traditional post-silicon debugging techniques that were described previously. However, the use of the ACE framework provides a promising post-silicon debugging tool that can ease, shorten, and reduce the cost of the post-silicon design process. The main advantages of ACE framework based post-silicon debugging are:

1. It eases the debugging process: ACE framework based debugging is very similar to the software debugging process, and therefore is trivial to understand and use by the debug engineer. This ease in debugging is achieved by providing complete accessibility and controllability of the hardware state to the debug engineer.

2. It can test potential design bug fixes without physically and permanently modifying the underlying hardware. This reduces both the cost and difficulty of post-silicon debugging by reducing the manual labor involved in fixing the design bugs.

3. It can accelerate the post-silicon debugging process because it does not require very slow procedures such as scan-out of the whole microarchitectural state or manual modification of the underlying hardware using the aforementioned FIB technique to evaluate potential design fixes.

5.3.2 ACE Framework Extensions for Manufacturing Testing

Manufacturing testing is the phase that follows chip fabrication and screens out parts with defective or weak devices. Today, most complex microprocessor designs use scan chains as the fundamental design for test (DFT) methodology. During the manufacturing testing phase, the design's scan chains are driven by external automatic test equipment

(ATE) that applies pre-generated test patterns to check the chip under test [23]. The test pattern set size depends on several factors, such as the design size, the fault models used, and the capabilities of the automatic test pattern generation (ATPG) tool used [45]. During the manufacturing testing phase, every single chip has to go through this testing process multiple times, at different voltage, temperature and frequency levels. Therefore, the manufacturing testing cost for each chip can be as high as 25-30% of the total manufacturing cost [45].

Motivation: Although this testing methodology served the semiconductor industry well for the last few decades, it has started to face an increasing number of challenges due to the exponential increase in the complexity of modern microprocessors [35], a product of the continuous silicon process technology scaling.

Specifically, the external ATE testers have a limited number of channels to drive the design's scan chains due to package pin limitations [45]. Furthermore, the speed of test pattern loading is limited by the maximum scan frequency that is usually much lower than the chip's operating frequency [45, 23]. The limited throughput of the scan interface between the external tester and the design under test constitutes the main bottleneck. These limitations, in combination with the larger set of test patterns required for testing modern multi-million gate designs leads to longer time spent on the tester per chip. Even today, the amount of time a chip spends on a tester can be several seconds [45]. Considering that the amortized testing cost of high-end test equipment is estimated to be at thousands of dollars per hour [18, 45], the conventional manufacturing testing process can be very cost-ineffective for microprocessor vendors.

Alternative Solutions: Logic built-in self-test (BIST) is a testing methodology based on pseudo-random test pattern generation and test response compaction. To speed up manufacturing testing, logic BIST techniques use the scan infrastructure to apply the on-chip pseudo-randomly generated test patterns and employ specialized hardware to compact the test responses [23]. Furthermore, the control signals used for testing are driven by an on-chip test controller. Therefore, a clear advantage of logic BIST over the traditional manufacturing testing methodology is that it significantly reduces the amount of data that is communicated between the tester and the chip. This leads to shorter testing times and, as a result, lower testing cost. Logic BIST also allows the manufacturing test to be performed at-speed (*i.e.*, at the chip's normal operating frequency rather than the frequency of the automatic test equipment), which improves both the speed and quality of testing.

Although logic BIST addresses major challenges of the traditional manufacturing testing methodology, it also imposes some new challenges. First, logic BIST requires the on-chip storage of a very large amount of pseudo-randomly generated test patterns. Sec-

ond, because logic BIST uses pseudo-randomly generated test patterns, it often provides significantly lower fault coverage than that provided by a much smaller number of high-quality, ATPG pre-generated test patterns [23]. Third, the use of the logic BIST methodology requires significantly more stringent design rules than conventional manufacturing testing [45]. For example, bus conflicts must be eliminated and the circuit must be made random-pattern testable [45]. Therefore, logic BIST techniques significantly increase both the hardware cost and the design complexity, while resulting in lower test coverage.

Proposed Solution - Use of the ACE Framework for Manufacturing Testing: The ACE infrastructure incorporates the advantages of both the scan-based and logic BIST testing methodologies, while it also can effectively address their limitations. Specifically, the ACE infrastructure provides two capabilities that are not together present in previous manufacturing testing techniques. First, the ACE framework is a built-in solution for fast loading of high-quality pre-generated ATPG test patterns into the scan-chain structures through software. This capability can eliminate the need for expensive and slow external equipment, currently needed for test pattern loading. Second, the ACE framework allows the test patterns to be loaded and applied at-speed at the chip's normal operating frequency rather than the much slower operating frequency of the automatic test equipment, which results in higher quality testing.

With these two capabilities, the ACE framework provides the best of both existing manufacturing testing techniques: 1) fast loading of test patterns to reduce testing time, 2) at-speed testing of the chip to improve testing quality as well as to reduce testing time, and 3) testing with ATPG pre-generated test patterns rather than the use of pseudo-randomly generated test patterns, to improve testing quality. Thus, if employed by the future integrated circuit manufacturing testing methodologies, it can greatly improve the speed, cost, and test coverage of the costly manufacturing testing phase of the microprocessor development cycle.

5.4 Chapter Summary

This chapter demonstrated that the ACE framework, presented in Chapter IV as a low-cost solution for online design bug detection and diagnosis, can be extended to other important applications to amortize its cost and ease its adoption in future generation microprocessor designs.

The first application that we considered as an ACE framework hardware extension is online design bug detection. We first described the problem of design bugs in modern microprocessors and motivated the need for the adoption of online design bug detection

mechanisms in future generation microprocessor designs. Next, we provided a rigorous analysis of processor design bugs in the RTL code of a commercial microprocessor, Sun's OpenSPARC T1 chip. Our low-level analysis of design bugs concluded that the signal monitoring requirements of online design bug detection are significantly higher than the estimates of previous studies. We believe that this discrepancy stems from the attempt in previous studies to infer low-level hardware implementation information from the high-level, abstract information provided in the microprocessor errata documents.

Based on the insights obtained from our rigorous design bug analysis, we proposed a novel distributed online bug detection mechanism based on the ACE framework. The proposed mechanism is able to flexibly monitor *all control signals*. This approach enables flexibility in bug detection because, unlike previous proposals, it does not rely on the successful selection of relevant signals at design time. Instead, any signal that can participate in the exercising of a bug can be monitored as needed.

In this chapter, we also described how the ACE framework can be extended to improve the quality and reduce the cost of two critical phases of microprocessor development: post-silicon debugging and manufacturing testing. Our descriptions showed that the flexibility provided by the ACE framework can significantly ease and accelerate the post-silicon debugging process by making the microarchitecture state easily accessible and controllable by the post-silicon debug engineers. Similarly, the flexibility of the ACE framework can eliminate the need for expensive automatic test equipment or costly yet lower-coverage hardware changes (*e.g.*, logic BIST) needed for manufacturing testing.

Finally, we evaluated the cost of the extended ACE framework on a detailed RTL prototype implementation and we found that the total silicon area overhead incurred is 15% of the whole OpenSPARC T1 chip, while the power consumption overhead is only 6.8%. Based on these numbers, it was demonstrated that the ACE framework is a general framework that can be used for multiple purposes to enhance the reliability and to reduce the design/testing cost of modern microprocessors and that it can provide additional value for its cost, something that would make its possible adoption in future generation microprocessors easier.

CHAPTER VI

FPGA-Based Accelerated Hardware Resiliency Analysis - The CrashTest Framework

A critical early stage in the development of a defect-tolerant microarchitecture is the assessment of the threats and the reliability requirements of the microprocessor design. During this process, system engineers employ hardware resiliency analysis tools to gauge the robustness of the microprocessor design and check if it meets the specified reliability targets. Hardware resiliency analysis tools are also useful to researchers for evaluating the effectiveness of existing and newly proposed microprocessor defect-tolerance techniques. The common approach followed by hardware resiliency analysis tools is to first inject faults in the microprocessor design and then analyze their impact on its behavior. After the fault injection and analysis process, the microprocessor design can be characterized for its reliability standards.

Today, simulation-based hardware resiliency analysis tools are limited by the use of high-level models of microarchitectural components that renders them incapable of faithfully modeling the silicon failure mechanisms. Furthermore, in order for current simulation-based resiliency analysis tools to gain statistical confidence over the generated results, the fault injection and analysis experiments need to be repeated several times in a Monte Carlo-like simulation environment that results in very long runtimes.

In order for hardware resiliency analysis tools to accurately gauge detailed circuit-level reliability phenomena and faithfully model silicon failure mechanisms, they need to use a detailed circuit-level model of the microprocessor. Design models that are capable of providing such a detailed circuit-level representation of the microprocessor are register-transfer level (RTL) models synthesized to gate-level netlists. However, the simulation of synthesized gate-level netlists in software is extremely slow, thus exacerbating the already long simulation runtimes.

This chapter presents CrashTest, a novel hardware resiliency analysis framework that addresses the challenges discussed above. Specifically, CrashTest automatically orchestrates a fault injection campaign and performs a detailed fault monitoring and analysis on the synthesized gate-level netlist of the design. Furthermore, the CrashTest framework is capable of accurately assessing the impact of run-time injected faults on the operation of large complex systems. In particular, in the CrsahTest framework the faults are injected into the design using novel gate-level logic transformations that instrument the design's netlist with fault emulation logic. The CrashTest framework is also augmented with a rich collection of fault models that encompass all variants of faults designers would expect to encounter at run time, ranging from soft faults to permanent silicon defects. The different fault models are defined by logic netlist transformations that can be easily modified and adapted by the user to model new failure mechanisms. Another novel characteristic of the CrashTest framework is that it employs FPGA-based accelerated hardware emulation to enable the analysis of complex full-system designs that can boot an operating system and run applications.

The remaining of this chapter is organized as follows: Section 6.1 discusses the challenges of accurate microprocessor resiliency analysis. Next, Section 6.2 gives a high-level overview of the CrashTest framework, while Sections 6.3 and 6.4 explain in detail the gate-level fault injection methodology and the FPGA-based fault emulation techniques used by the CrashTest framework. Section 6.5 evaluates the performance of CrashTest and presents experimental results that demonstrate its application and effectiveness, while Section 6.6 briefly describes related previous work. Finally, the work presented in this chapter is summarized in Section 6.7.

6.1 The Challenges of Hardware Resiliency Analysis

The process of accurately assessing the robustness of a hardware design or evaluating the effectiveness of a fault-tolerant technique, places some challenging set of requirements on the hardware resiliency analysis infrastructure.

- ***Low-level Fault Analysis***: High fidelity is a very important aspect of a hardware resiliency analysis framework. Using high-level models of microarchitectural components with limited knowledge of the underlying circuit is inadequate to perform high-fidelity resiliency analysis. In order to correctly model the introduction, propagation, and possible masking of the faults, the hardware resiliency analysis framework must accurately gauge circuit-level phenomena using a detailed low-level model of the design under analysis (*e.g.*, gate-level netlist).

- ***Flexible Fault Modeling***: Due to the existence of multiple silicon reliability threats, the resiliency analysis framework needs to support an extensive collection of low-level fault models to cover silicon failure mechanisms that range from transient faults, to manufacturing faults, process variation induced faults, and silicon wear-out related faults. Moreover, fault modeling is an open area of research with continuous advancements [23, 40]. Often, new fault models are devised targeting emerging silicon failure modes or more accurately modeling existing failure mechanisms. Therefore, it is crucial that the fault model collection of a hardware resiliency analysis framework can be easily upgraded with new fault models.

- ***Fast Design Simulation***: The simulation of the design must deliver sufficient performance to enable the analysis of complex systems, including booting an operating system and running applications. This will enable users to assess the impact of faults at the full system and application level and still have a quick turnaround for the evaluation.

- ***Flexible Simulation Interface***: It is critical for the usability of the hardware resiliency analysis framework to provide an intuitive way to analyze a wide range of hardware designs and fault-tolerant techniques. To this end, the resiliency analysis framework demands a flexible interface and proper stubs to accommodate the evaluation of different systems.

Given the challenging set of requirements for hardware resiliency analysis, the CrashTest framework is focused toward the use of fault injection campaigns performed at the gate-level model, accelerated by FPGA-based hardware emulation in order to achieve both accuracy and performance.

6.2 Overview of the CrashTest Framework

The goal of the CrashTest hardware resiliency analysis framework is to provide a fast, high-fidelity, and comprehensive analysis of the effects of several different fault models on the applications running on the design under analysis (this could be either an unprotected design or a fault-tolerant design). Given the specification of the design under analysis in a hardware description language (HDL), CrashTest automatically orchestrates a fault injection/analysis campaign. This process is composed of two stages: (i) the front-end translation that generates the fault-injection ready gate-level netlist of the design under analysis, and (ii) the back-end fault simulation and analysis that performs the actual fault

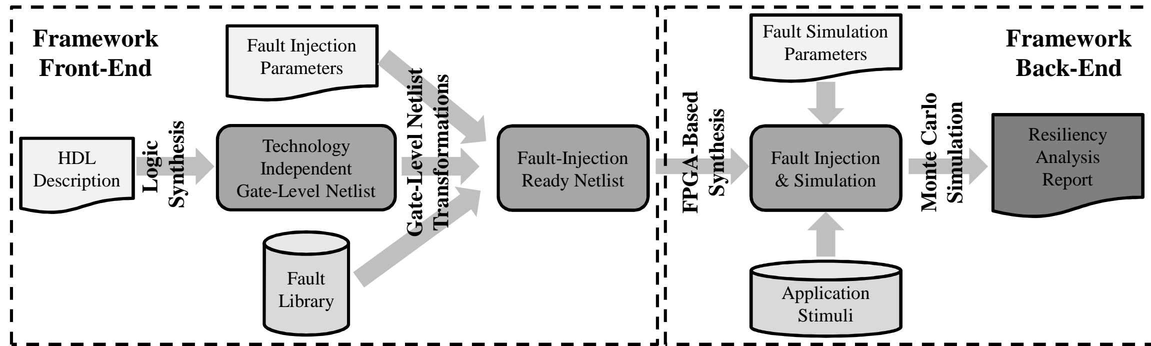


Figure 6.1: Overview of the CrashTest Hardware Resiliency Analysis Framework: The framework is composed of (i) the front-end stage generating the fault injection-ready gate-level netlist and (ii) the back-end stage performing fault injection and analysis and generating the final resiliency analysis report.

injection and fault monitoring and evaluates the effects of the injected faults. The overview of this process is illustrated in Figure 6.1.

Framework Front-End: First, the HDL model of the design under analysis is provided by the user (either in Verilog or VHDL). Subsequently, the HDL model of the design is synthesized by the front-end stage of the framework using a technology-independent standard cells library to get a *technology-independent gate-level netlist* of the design. For each standard cell in the library (*i.e.*, a combinational gate or a sequential element), CrashTest is enhanced with a *gate-level logic transformation* that can modify the netlist and insert extra fault injection logic. This extra logic can be activated at runtime to emulate the effects of a fault injected into the cell. We developed a wide range of fault models and gate-level logic transformations to provide the capability of emulating different failure mechanisms. The collection of all logic transformations is stored in the framework's *fault library*. Based on the *injection parameters* selected by the user (*i.e.*, the fault models and the injection locations), the framework automatically generates the *fault injection-ready netlist* of the design using the logic transformations in the library. This netlist is then delivered to the fault analysis simulator at the back-end stage.

Framework Back-End: At the framework back end, the fault injection-ready netlist is re-synthesized and mapped on an FPGA. At this point the fault injection and analysis campaign is ready to begin. Based on the *fault simulation parameters* given by the user, the fault injection/analysis emulator injects faults at different sites in the netlist and monitors their propagation and impact on the design and the running applications. During fault emulation, the design under analysis is exercised with the *application stimuli*. To gain statistical confidence on the provided results, the experiments are repeated in a Monte Carlo simulation model by altering the fault sites and/or the application stimuli. After

running a sufficient number of experiments to gain statistical confidence, the results are aggregated into the *resiliency analysis report* which is the final deliverable of the CrashTest framework.

In the following sections, we describe each step and each process of the CrashTest framework in more detail.

6.3 Gate-Level Fault Injection Methodology

Technology Independent Logic Synthesis - The first step in the front-end stage of the CrashTest framework is to convert the user-provided high-level HDL model of the design under analysis into a common format that the framework can analyze and get an accurate list of candidate circuit locations to perform gate-level fault injection. This is achieved by performing logic synthesis with Synopsys Design Compiler targeting a technology-independent standard cell library (GTECH). The resulting gate-level netlist is composed of simple logic gates (*e.g.*, AND, OR, NOT, Flip-Flops, *etc.*) and it is free from any fabrication technology related characteristics and properties. This gate-level netlist is subsequently parsed to generate a list of all possible fault injection locations in the circuit (*i.e.*, a list of all logic gates and flip-flops in the design). This list is used by the user to specify the fault injection locations. Alternatively, if randomized fault injection is desired, random selection of fault sites can be performed by the framework.

Netlist Fault Injection Instrumentation - Once fault locations are selected, the gate-level netlist is instrumented with extra fault injection logic that, when enabled, emulates the effects of the injected faults. Each fault model supported by the framework is associated with a gate-level logic transformation that modifies the netlist and instruments it with the extra fault injection logic. The collection of gate-level logic transformations composes the framework's fault library. This modular design makes it fairly easy to upgrade the framework with new fault models by simply implementing and adding new netlist logic transformations into the fault library.

Fault Models - The CrashTest hardware resiliency analysis framework is already enhanced with a collection of fault models and their corresponding netlist logic transformations. This fault model collection covers an extensive spectrum of silicon failure mechanisms ranging from transient faults due to cosmic rays to permanent faults due to silicon wearout:

- **Stuck-at:** The stuck-at fault model is the industry standard model for circuit testing. It assumes that a circuit defect behaves as a node stuck at logical 0 or 1. The stuck-at

fault model is most commonly used to mimic permanent manufacturing or wearout-related silicon defects.

- **Stuck-open:** The stuck-open fault model assumes that a single physical line in the circuit is broken. The unconnected node is not tied to either Vcc or Gnd and its behavior is rather unpredictable (logical 0 or 1 or high impedance). The stuck-open fault model is commonly used to mimic permanent defects that are not covered by the stuck-at fault model.

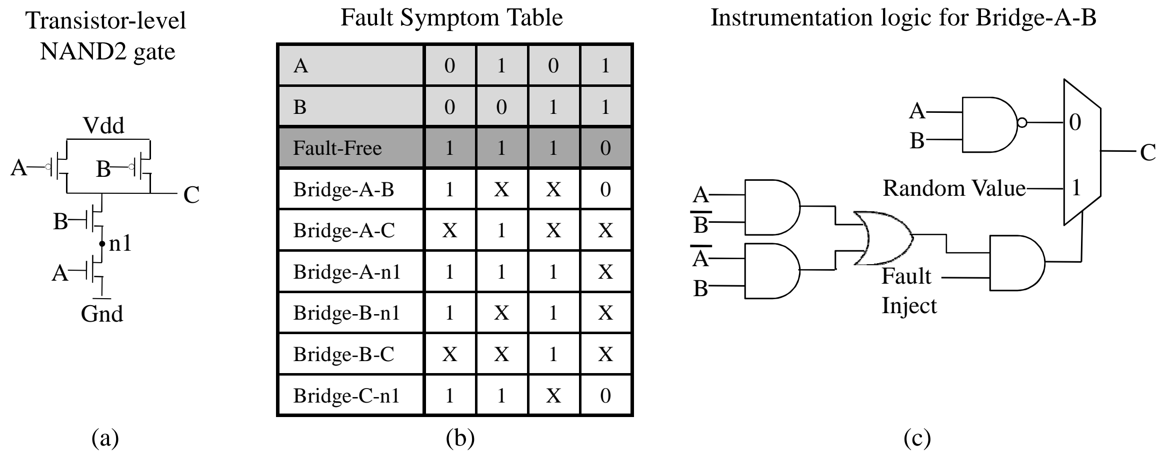
- **Bridge:** The bridge fault model assumes that two nodes of a circuit are shorted together. The behavior of the two shorted nodes depends on the values and the strength of their driving nodes. The bridge fault model covers a large percentage of permanent manufacturing or wearout-related defects.

- **Path-delay:** The path-delay fault model assumes that the logic function of the circuit is correct, however, the total delay in a path from its inputs to outputs exceeds the predefined threshold and it causes incorrect behavior. The path-delay fault model is most commonly used to mimic the effects of process variation or device degradation due to age-related wearout.

- **Single Event Upset:** The single event upset (SEU) fault model assumes that the value of a node in the circuit if flipped for one cycle. After this one cycle upset, the node behaves as expected. The SEU fault model is used to mimic transient faults that are most commonly used by cosmic radiation or alpha particles.

Gate-Level Logic Transformations - Some fault models require trivial gate-level logic transformations. For example, the instrumentation needed to emulate a stuck-at fault is just a multiplexer that controls the output of the faulty gate and has one of its inputs connected to logic zero/one. However, there are fault models that are more complex and affect the design at the transistor level. For example, the bridge fault model assumes that two nodes in the design are shorted together. To emulate the effect of a bridge fault model with high fidelity, we simulated the faulty gates at the CMOS transistor level and generated the corresponding *fault symptom tables*. To illustrate this process, Figure 6.2(a) shows the CMOS transistor level representation of a NAND2 logic gate, while Figure 6.2(b) shows the respective fault symptom table of the bridge fault model.

By observing the fault symptom table we notice that for some inputs the effects of the fault are masked, thus the faulty gate behaves exactly like a fault-free gate. However, for other input combinations the fault's effects propagate to the gate's output and result into



A	0	1	0	1
B	0	0	1	1
Fault-Free	1	1	1	0
Bridge-A-B	1	X	X	0
Bridge-A-C	X	1	X	X
Bridge-A-n1	1	1	1	X
Bridge-B-n1	1	X	1	X
Bridge-B-C	X	X	1	X
Bridge-C-n1	1	1	X	0

Figure 6.2: Logic Transformations - Bridge Fault: The CMOS transistor-level design of a gate in (a) is used to generate the gate's fault symptom table for the bridge fault model that is shown in (b). Part (c) shows the instrumentation logic for emulating the effects of the Bridge-A-B fault.

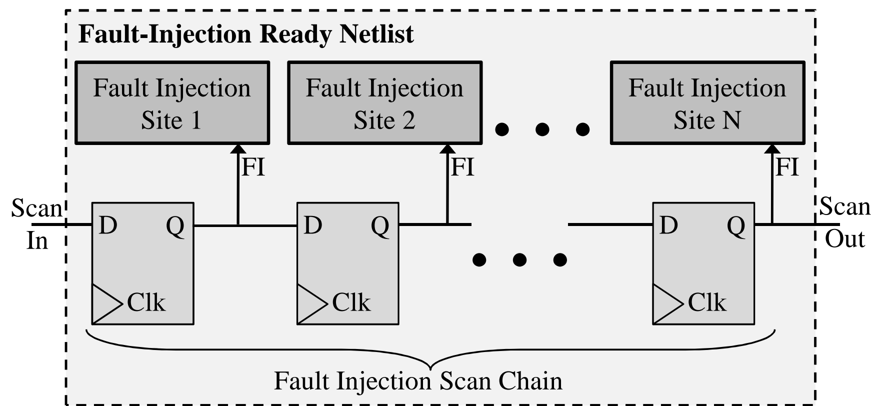


Figure 6.3: Fault Injection Scan Chain: The netlist is instrumented with fault injection logic for multiple faults. The scan chain controls the enabling of the injected faults during emulation.

an unstable output signal that could be either a logic zero or one (Random Value in Figure 6.2(c)). The framework's fault library is populated with a fault symptom table for each combination of a standard cell library gate and a supported fault model. Given the gate type and the fault model, the netlist instrumentation routine accesses the fault library and applies the respective logic transformation that would insert the necessary instrumentation logic to emulate the fault effects. Figure 6.2(c) shows the instrumentation logic needed to emulate the effects of a bridge fault between the circuit nodes A and B of the NAND2 gate.

Fault Injection Scan Chain - To avoid re-instrumenting the netlist each time a new fault is injected and simulated, the netlist can be instrumented for multiple faults at multiple locations. This accelerates the fault emulation at the back-end of the framework, but

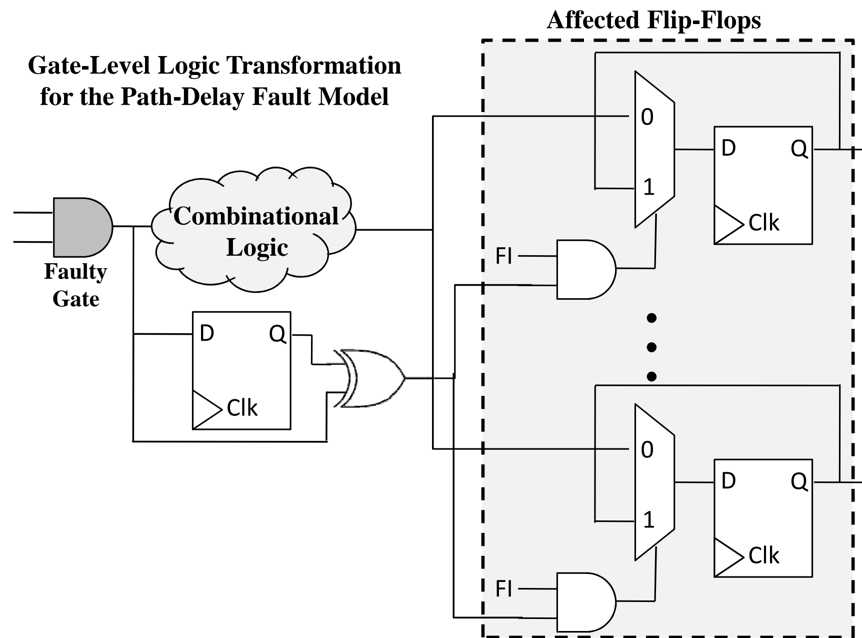


Figure 6.4: Logic Transformation for the Path-Delay Fault Model: If the output of the faulty gate changes in a given cycle, all affected flip-flops miss latching the newly computed value and hold the previous cycle's value.

it also increases the instrumented circuit size. The insertion of each fault into the netlist also adds an extra control signal used for enabling and disabling the inserted fault at run-time (for instance, signal *Fault Inject* and *Random Value* in 6.2(c)). During emulation, these signals are accessible by the *Fault Injection Manager* (see Section 6.4) through a fault injection scan chain. This scan chain is automatically inserted during the netlist instrumentation phase and it greatly simplifies the interface between the injection interface and the emulated faulty design. The number of faults that can be instrumented using this method is arbitrary and it is limited only by the size of the target FPGA device. The design of the fault injection scan chain is illustrated in Figure 6.3.

The Path-Delay Fault Model - The gate-level logic transformations employed for the rest of the supported fault models are similar to the one presented at Figure 6.2(c) for the bridge fault. One exception is the path-delay fault model which has slightly different characteristics. Path-delay faults are characterized by slower combinational logic gates that cause longer path delays than the ones expected at design time. Whenever these slower gates get exercised, they can increase the path delay beyond the critical path delay and cause timing violations (*i.e.*, the flip-flops at the end of the path miss to latch the newly computed value). In our framework, the effects of the path-delay fault model are emulated by the gate-level logic transformation shown at Figure 6.4. To find out the set of flip-flops that are affected by the slower *faulty gate*, we trace forward the combinational logic and

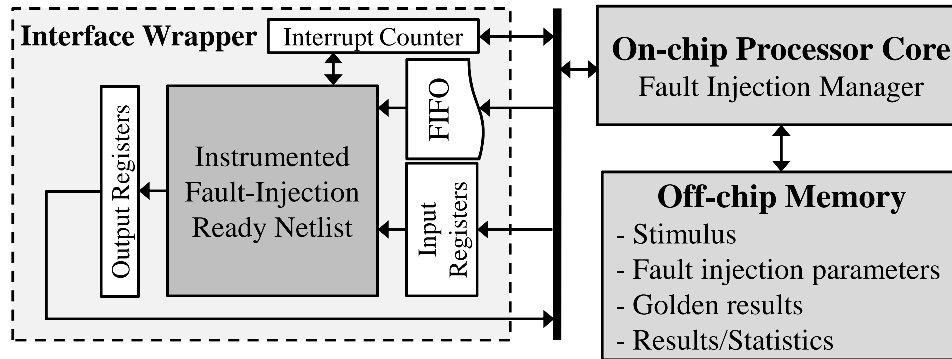


Figure 6.5: FPGA-Based Fault Injection and Simulation: The FPGA-mapped netlist is wrapped by a standard interface providing a seamless connection to the fault injection manager that is running on an on-chip processor core. All experiment data and results are stored on an off-chip memory.

find all those flip-flops that have a path that includes the faulty gate. From that set of flip-flops we choose only those that have a path delay with a timing slack smaller than a predefined threshold specified by the user (*i.e.*, the expected delay due to the faulty gate).

6.4 FPGA-Based Fault Emulation

CrashTest employs an FPGA platform to emulate the fault injected hardware and accelerate the fault simulation and analysis process. The first step in this process it to synthesize and map the fault injection-ready netlist to the target FPGA. To provide a standard simulation interface that is independent of the design under analysis, we add an automatically generated *interface wrapper* to the fault injected-ready netlist. This interface wrapper provides a seamless connection with the *fault injection manager*, which is an automatically generated software program responsible for orchestrating the fault injection and analysis process. The interface wrapper and the fault injection manager are connected through an *on-chip interconnect bus*. Figure 6.5 shows the major components and the data-flow of the fault injection, simulation and analysis process.

Fault Injection Manager - During the emulation and analysis process, the FPGA-mapped design is exercised and controlled by the fault injection manager. In our experiments we used a Xilinx Virtex-II Pro FPGA, which has two on-chip PowerPC processors, with the fault injection manager software running on one of them. Alternatively, the fault injection manager can also run on a soft-core (*e.g.*, Microblaze). Specifically, the fault injection manager is responsible for the following tasks:

- Feed the instrumented injection scan chain with all the control signals required to perform the fault injection campaign. This is done through a FIFO queue updated

whenever a new fault is injected into the design. The fault injection parameters (*i.e.*, fault location and time) are stored on an off-chip memory accessible by the fault injection manager.

- Stimulate the FPGA-mapped design through the input registers. The applications stimulus is either provided by the user or automatically generated, and it is stored in the off-chip memory.
- Monitor the output of the FPGA-mapped design for errors through the output registers. The output is compared to a golden output that is collected through a fault-free version of the same design and it is stored in the off-chip memory.
- Maintain fault analysis statistics and store the results to the off-chip memory for later processing.
- Synchronize the FPGA-mapped design with the fault injection and analysis process through the interrupt counter.

6.5 Framework Evaluation

In this section, we evaluate our FPGA-based resiliency analysis infrastructure and compare its performance to an equivalent software-based implementation. In addition, we perform an initial study by using the CrashTest infrastructure to examine the effects of design resiliency as the underlying fault models are changed.

6.5.1 Experimental Methodology

Benchmark Designs - For the evaluation of CrashTest we used three benchmark designs. These benchmark designs and their characteristics are shown at Table 6.1. The chip-multiprocessor (CMP) interconnect router implements a wormhole router pipelined at the flit level with credit-based flow control functionality for a two-dimensional torus network topology [100]. We used SPEC CPU2000 communication traces derived from the TRIPS architecture [108] to provide application stimuli to the router. The DLX core is a 32-bit 5-stage in-order single-issue pipeline running the MIPS-Lite ISA. Finally, the LEON3 is a system-on-chip including a 32-bit 7-stage pipelined processor running the SPARC V8 architecture, an on-chip interconnect, basic peripherals and a memory controller [37] able of booting an unmodified version of Linux 2.6. The LEON processor was configured without on-chip caches and faults were injected only in the core component.

Benchmark Name	Logic Gates (GTECH)	Flip Flops	Description
CMP Router	16,544	1,705	chip-multiprocessor interconnect router for a 2D mesh network with 32-bit flits
DLX Core	15,015	2,030	5-stage in-order DLX pipeline running MIPS-Lite ISA
LEON3 System-on-chip	66,312	6,925	System-on-chip with a 7-stage pipeline 32-bit processor compliant with the SPARC V8 architecture, an on-chip interconnect, basic peripherals and a memory controller.

Table 6.1: Benchmark Designs: Characteristics of the benchmark designs used to evaluate the CrashTest framework.

Netlist Fault-Injection Instrumentation - The HDL model of the design under analysis is synthesized using the Synopsys Design Compiler and the GTECH standard cell library. The resulting netlist is a technology-independent GTECH gate-level netlist. The gate-level netlist is subsequently parsed by Perl scripts to locate all the possible injection sites in the circuit. Once the sites and fault types are selected (using a uniform random distribution for these experiments), a Perl script implements gate-level logic transformations to instrument the netlist with the necessary fault injection logic.

Software-Based Analysis Methodology - The software-based fault simulation and analysis is performed using the Synopsys VCS logic simulator for the CMP router and the DLX core. For the simulation of the LEON3 system-on-chip we used ModelSim since it required the simulation of both Verilog and VHDL modules. The fault simulations using VCS were run on an Intel Core 2 Duo running at 2.13GHz with a 2MB L2 cache and 2GB of RAM, while the ModelSim simulations were run on a P4 at 3.4GHz and 2GB RAM.

FPGA-Based Analysis Methodology - For the FPGA-based fault emulation and analysis we used the XUP V2P Development Board [148]. The board is equipped with a Virtex-2 Pro XC2VP30 FPGA with 13,696 slices (each with two 4-input LUTs and two flip-flops), and two PowerPC 405 processors. At the time of writing, this FPGA represented a mid-sized device; devices with up to 10X as many resources are currently available. For off-chip memory we used one 256MB module of DRAM. The main tools used to develop the CrashTest framework are the Xilinx Platform Studio (XPS) version 9.1i in combination with Xilinx Integrated Software Environment 9.1i (ISE). We also used Synplicity's Synplify 9.0.1 for the FPGA-based synthesis. The FPGA synthesis and mapping process was ran on a P4 CPU at 3.0Ghz and 1GB RAM. The synthesis and mapping process for the LEON3 system took about 45 minutes, while the other two benchmark designs required significantly less time.

Confidence Level = 95%	Confidence Interval		
Number of Fault Injections (Sample Size)	**CMP Router (18249 gates)**	**DLX Core (17045 gates)**	**LEON3 (73237 gates)**
256	±6.08	±6.08	±6.11
512	±4.27	±4.27	±4.32
1024	±2.98	±2.96	±3.04
2048	±2.04	±2.03	±2.14
4096	±1.35	±1.33	±1.49
8192	±0.8	±0.78	±1.02

Table 6.2: Statistical Confidence: The Table shows the confidence level of the results obtained when different number of faults are injected during the injection campaigns for our benchmark designs.

6.5.2 Monte Carlo Simulation & Statistical Confidence

Performing gate-level fault injection campaigns in complex designs and observing their impact at the application level is a fairly computationally intensive process. The propagation of fault effects from the gate level to the application level requires a significant amount of gate-level simulation of the design under analysis. A common practice used to reduce the number of fault injections and make the resiliency analysis process more computationally tractable is the use of Monte Carlo simulation methods. Through Monte Carlo simulation, fault injection experiments are repeated by randomly changing the fault injection location and time (*i.e.*, the clock cycle that the fault will be enabled). The number of times that the Monte Carlo experiments are repeated depends on the desired statistical confidence that will characterize the obtained results.

Table 6.2 shows the confidence intervals for different numbers of fault injection experiments for the three benchmark designs. These figures were calculated using the statistical sample size formulas from [10]. For most applications, a confidence level of 95% and a confidence interval of 3% are acceptable. From Table 6.2 we notice that this degree of statistical confidence can be achieved by 1024 fault injections for all three benchmark designs.

6.5.3 Framework Performance

Fault Injection Logic Overhead - Table 6.3 shows the allocated FPGA resources when the baseline (fault-free) benchmark designs were synthesized and mapped on the FPGA. When the designs are augmented with the fault simulation interface wrapper the utilization of the FPGA slices is increased from 15% to 31%. As shown in the fourth and fifth columns of Table 6.3, not all of the flip-flops and LUTs in each utilized slice

Bench. Design	Injected Faults	Slices (out of 13696)	Slice Flip Flops (out of 27392)	4 Input LUTs (out of 27392)
CMP Router	0 (baseline)	2968 (21%)	3021 (11%)	3705 (13%)
	0 (wrapper)	6679 (48%)	4731 (17%)	10840 (39%)
	8	6718 (49%)	4745 (17%)	10781 (39%)
	64	6912 (50%)	4857 (17%)	11192 (40%)
	128	7161 (52%)	4985 (18%)	11408 (41%)
	256	7279 (53%)	5241 (19%)	11425 (41%)
	512	7854 (57%)	5753 (21%)	12020 (43%)
	1024	8903 (65%)	6778 (24%)	13059 (47%)
DLX Core	0 (baseline)	2499 (18%)	2520 (9%)	2386 (8%)
	0 (wrapper)	6820 (49%)	8202 (29%)	4573 (16%)
	1024	9593 (70%)	6700 (24%)	9948 (36%)
LEON3 System-on-chip	0 (baseline)	10281 (75%)	10178 (37%)	20562 (75%)
	0 (wrapper)	11057 (80%)	11103 (40%)	22113 (80%)
	1024	11785 (86%)	13146 (47%)	23570 (86%)

Table 6.3: Fault Injection Logic Overhead: Utilization of the FPGA resources comparing the baseline (fault-free) designs and the fault injection instrumented designs mapped on the FPGA.

are used. The table also shows the overhead of the instrumentation logic for designs injected with different numbers of stuck-at faults. The capability of injecting several faults into the design is very important since it significantly accelerates the fault simulation process by avoiding time-consuming iterations of netlist instrumentation and FPGA synthesis/mapping.

Fault Simulation/Analysis Speed - Table 6.4 compares the speed of the software-based and the FPGA-based fault emulation and analysis engines. For the CMP router design we noticed that the speed of the software-based scheme varied for different fault models. This difference stems from the different logic complexity required to emulate the behavior of each fault model. On average, for the CMP router the software-based scheme provides a simulation speed that is in the order of 10 KHz. We have observed similar results for the DLX core design (not shown in the table for brevity). On the other hand, the speed of the FPGA-based scheme is not affected by the fault injection logic. Therefore, all fault models are emulated with the same clock frequency and achieve the same emulation speed. For the CMP router, the speed of the emulation framework is 220 KHz, leading to an average speed up of $\approx$20X for simple fault models and $\approx$85X for the more complex fault models.

The simulation speed achieved by the software-based scheme when analyzing the LEON3 system-on-chip is much lower than the one observed for the other two simpler designs (*i.e.*, the CMP router and the DLX core). Specifically, the simulation speed is

<table>
<tr><th>Bench. Design</th><th>Fault Model</th><th>Software-Based Fault Simulation Speed</th><th>FPGA-Based Fault Simulation Speed</th><th>Speed Up</th></tr>
<tr><td rowspan="6">CMP Router</td><td>Stuck-at-0</td><td>9.75 KHz</td><td rowspan="6">220 KHz</td><td>22X</td></tr>
<tr><td>Stuck-at-1</td><td>8.09 KHz</td><td>27X</td></tr>
<tr><td>Stuck-open</td><td>2.42 KHz</td><td>90X</td></tr>
<tr><td>Bridge</td><td>2.63KHz</td><td>83X</td></tr>
<tr><td>Path-delay</td><td>11.34 KHz</td><td>19X</td></tr>
<tr><td>SEU</td><td>13.04 KHz</td><td>16X</td></tr>
<tr><td>LEON3 System-on-chip</td><td>Stuck-at-0</td><td>28 Hz</td><td>25 Mhz</td><td>~900 000X</td></tr>
</table>

Table 6.4: Fault Simulation Speed: Performance comparison of the software- and FPGA-based fault simulation engines.

limited to 28 Hz, due to the much higher complexity of the full-system LEON3 design. In contrast, the emulation speed of the LEON3 system on the FPGA-based scheme is faster than the other two simpler designs. This is due to how the application stimulus is applied to different designs by the fault injection manager. Since the LEON3 full-system design includes a memory controller, the interaction with the external environment is limited to memory read/write requests, which are serviced by the off-chip DRAM module. Therefore in the LEON3 analysis there is very little interaction between the fault injection manager and the design under analysis in feeding the application stimulus. On the other hand, when emulating the other two designs, the fault injection manager must provide input stimuli cycle-by-cycle in order to drive the emulation, thus limiting the overall performance. The emulation speed of the LEON3 design on the FPGA-based scheme is 25 MHz, which leads to a six orders of magnitude speedup compared to the corresponding simulation speed achieved by the software-based scheme.

6.5.4 Experimental Results

Fault Effects per Fault Model - The graph in Figure 6.6 shows the percentage of injected faults that caused a failure, grouped by fault model. The fault injection experiments were run on the CMP router stimulated with communication traces of several SPEC CPU2000 benchmarks and a synthetic high-traffic communication trace (*hi_util*). We observe that the effects of the injected faults on the design vary for different fault models. Specifically, fault models of permanent silicon failures (*i.e.*, stuck-at, stuck-open, and bridge) have more adverse effects on the design, and 70-80% of them cause an error that is observable at the primary outputs of the design during the emulation. On the other hand, the path delay fault model has less adverse effects, and on average only 40% of these faults

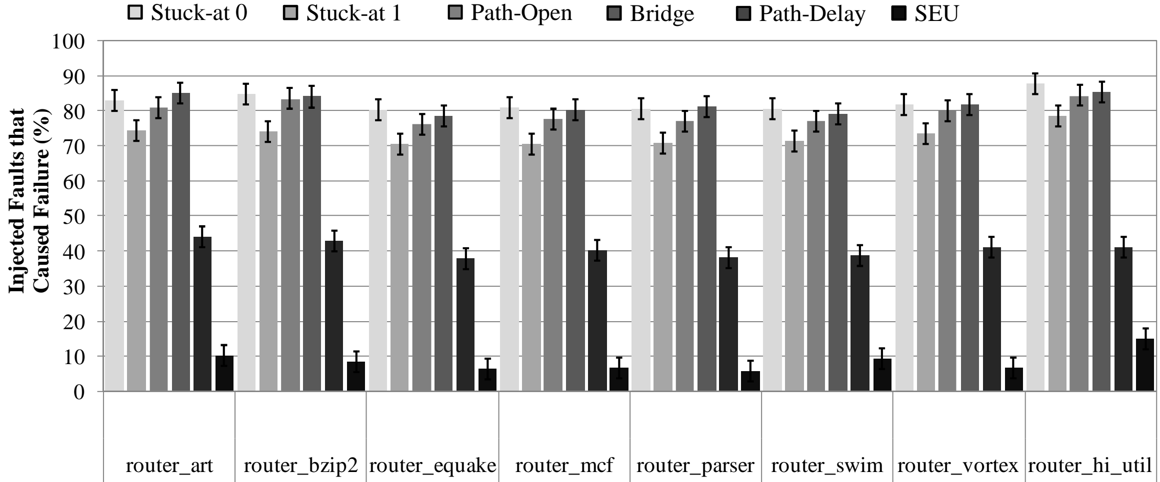


Figure 6.6: Design Resiliency vs. Underlying Fault Model: Percentage of injected faults that were exposed for each fault model. Experiments are run on the CMP router using SPEC2000 traces.

manifest an error. Finally, the SEU faults have the least impact on the correct functionality of the design and on average less than 10% of them cause an error.

Failure Observation Latency - The graph in Figure 6.7 shows the average latency of an injected fault to propagate an error to the primary outputs of the design. The results shown are for different fault models for the CMP router and the LEON3 system-on-chip. The failure observation latency is a very important metric when assessing the resiliency of a design because it provides insight on whether specific error detection and recovery techniques can provide a detection and recovery window that would allow a successful recovery from the fault's effects. We notice that the failure observation latency varies depending on the fault model. Specifically, we observe that for the CMP router the injected path-delay faults have the highest failure manifestation latency, while fault models associated with permanent failure mechanisms usually have similar failure manifestation latencies. Furthermore, we notice that the error manifestation latency for SEU faults is very small. When this observation is combined with the results of the previous experiment, we conclude that SEU transient faults either cause an error in the design immediately after they occur, or they do not cause an error at all, as would be expected due to their transient nature.

We also notice that the measured failure observation latencies for the LEON3 system-on-chip are orders of magnitude larger than the ones observed for the CMP router. This difference stems from the higher complexity of the LEON3 system-on-chip which leads to more cycles required for a fault to propagate to the design's output (the output of the running application). To give more insights regarding the failure observation latency of the faults injected in the LEON3 system, the graph of Figure 6.8 shows the cumulative

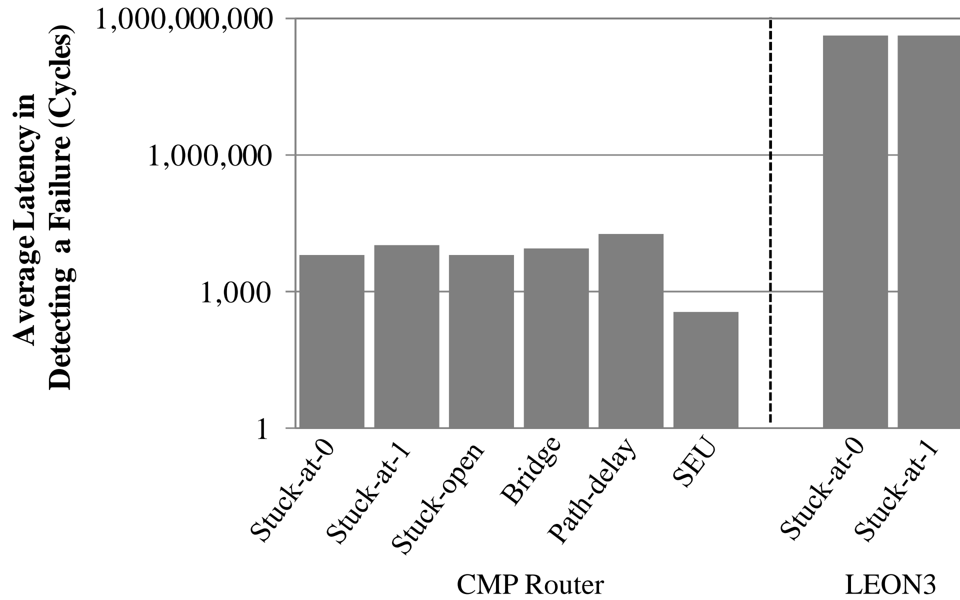


Figure 6.7: Failure Detection Latency: Failure observation latency at the design's primary outputs.

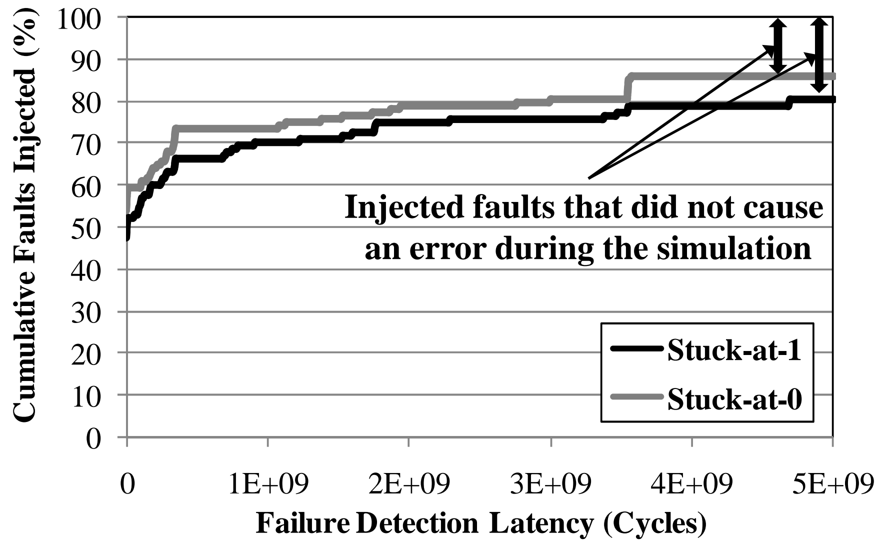


Figure 6.8: Application-Level Detection Latency: Latency (in cycles) for a stuck-at fault to propagate to the application results in the LEON3 SoC.

distribution of the injected faults over the failure observation latency in clock cycles. An interesting observation is that more than half of the injected faults propagate a failure to the application output almost immediately, but the remaining ones require billions of cycles for the failure to manifest. This observation supports the argument that if a fault hits a critical part of the design, then its effect are immediate. On the other hand, if it hits a less critical/exercised part of the design, then its effects are delayed by long latencies.

6.6 Related Work

Fault Simulation vs. Resiliency Analysis: Fault simulators are software tools that can determine the set of faults that can be exposed by a given test vector. They are mainly used for ATPG (Automatic Test Pattern Generation) with the objective of measuring the fault coverage of a given set of test vectors [23]. On the other hand, resiliency analysis tools employ fault injection campaigns on a design executing typical workloads to measure the impact that the injected faults have on the design's operation and on the running applications. Although both methodologies use fault models to simulate the effects of faults on the circuit under test, their goals and requirements are fundamentally different.

For example, fault simulators need to simulate the design under test only for a limited number of clock cycles to grade the test vectors. Furthermore, in order to measure the fault coverage of the test vectors, fault simulators need to activate a fault in every single node in the design. In contrast, resiliency analysis tools need to simulate the design under analysis for a significant amount of clock cycles in order to observe the fault effects at the application level. Moreover, resiliency analysis tools usually employ Monte Carlo simulation methodologies and inject only the number of faults required to provide adequate statistical confidence for the results obtained. Due to these key different characteristics of the two methodologies, ATPG fault simulators cannot be efficiently used as a fault injection substrate to perform design resiliency analysis.

Several works in the literature have proposed resiliency analysis frameworks that are based on fault injection campaigns. These works can be partitioned into software-based and hardware-based resiliency analysis, based on the methodology used to perform the fault simulation and analysis [84].

Software-Based Resiliency Analysis: Often, software-based fault injection is preferred to hardware-based solutions due to its low cost, faster and less complex development cycle, flexibility of customization, or simply because no low-level hardware model of the design is available. There are several software-based resiliency analysis frameworks presented in the literature [62, 107, 142]. Although they have many advantages, the major limitation of software-based fault injection is that it is too slow to perform low-level (*e.g.*, gate-level) fault simulation and analysis on complex designs or full systems running software applications. One way to address this issue is by using high-level models of a design (*i.e.*, microarchitectural models), but this higher level of abstraction and the lack of circuit-level information jeopardizes the fidelity of the resiliency analysis results. Another workaround is to limit the complexity of the design under analysis down to blocks of a few thousands gates, but this greatly limits the usability of the approach.

Hardware-Based Resiliency Analysis: The performance limitation of the software-based fault injection approach can be addressed by employing hardware-based fault injection. Hardware-based resiliency analysis frameworks usually employ FPGAs (Field Programmable Gate Arrays) that are capable of emulating the fault injected design orders of magnitude faster than software-based approaches, therefore significantly speeding up the fault simulation and analysis process. Although the use of FPGA emulation platforms addresses the limited performance of the software frameworks, it introduces some other major challenges. Specifically, by employing FPGA platforms to emulate the fault injected design, the automation of the fault injection and analysis process becomes more challenging. Furthermore, FPGA-based resiliency analysis frameworks are characterized by the difficulty of mapping complex fault models into hardware which greatly limits the range of supported fault models. Hence, the previously proposed hardware-based resiliency analysis frameworks were limited to simple transient and stuck-at faults [27, 76].

6.7 Chapter Summary

This chapter presented CrashTest, a novel FGPA-based resiliency analysis framework capable of automatically orchestrating a fault injection and analysis campaign on the gate-level netlist of the design. To accelerate the fault injection process, multiple faults are injected into the design simultaneously by instrumenting the netlist with fault injection logic through gate-level logic transformations. The CrashTest framework supports an extended collection of fault models ranging from transient faults to silicon defects, and it can easily be upgraded with new fault models. In addition, the CrashTest framework employs FPGA-based accelerated hardware emulation to enable the analysis of complex full-system designs that can boot an operating system and run applications.

The CrashTest hardware resiliency analysis framework was evaluated on a commercial FPGA-based platform and we found that the use of hardware emulation, when compared to an equivalent software-based hardware resiliency analysis simulator, it can accelerate the fault simulation and analysis process by 16-90x for simple designs and six orders of magnitude for a more complex system-on-chip design.

CHAPTER VII

Conclusions and Future Work

Silicon process technology scaling has been one of the major driving forces in the impressive growth of the semiconductor industry for several decades. The continuous silicon process technology scaling over these decades offered smaller, faster, and cheaper transistors to the microprocessor manufacturers that enabled the development of more powerful and cheaper microprocessors. The concurrent development of more capable and cheaper microprocessors with that of more advanced and easier to use software, flooded our society with microprocessor-based electronic products with applications that touch every aspect of our life.

However, as silicon process technology scales into extremely small transistor sizes, with dimensions that measure in just few atoms, new challenges have developed in maintaining transistor reliability and offering a reliable fabrication substrate that will guarantee durable microprocessor designs. As argued in Chapter I, many technology experts today warn that we are reaching the limits of what traditional silicon scaling can achieve and that we are entering an era where any further silicon process technology scaling will have major effects on transistor reliability. This has a strong implication on the design of future generation microprocessors: indeed, the durability and widespread use of microprocessors relies on highly reliable silicon processes exhibiting very low failure rates. However, as the reliability wanes, new design paradigms will need to be developed and adopted that allow to fabricate reliable systems out of unreliable devices. This will entail adopting new design techniques to tolerate silicon defects that might occur during the lifetime of the microprocessor design and still present high reliability standards to the end user.

As discussed in Chapter II, although today high-end computing systems for critical applications demanding high reliability standards are already augmented with defect-tolerance techniques, these techniques incur high overheads and account for a significant fraction of the microprocessor's area and power consumption budgets. To this extent, this thesis makes the case that novel and clever defect-tolerance techniques can offer to future

generation microprocessor designs the same reliability guarantees at a much lower cost, so to enable the adoption of reliability solutions in mainstream, cost-sensitive microprocessor designs.

7.1 Thesis Summary

This thesis developed low-cost defect-tolerance solutions that can provide to microprocessor designs the same reliability guarantees as those of costly traditional defect-tolerance techniques that today are deployed only in high-end systems. To make this possible, this thesis suggested a paradigm shift in the way the defect tolerance is provided to microprocessor designs. Specifically, traditional defect-tolerance techniques saddle the microprocessor design with extra hardware components that continuously monitor the execution for errors through redundant computation. These redundant hardware resources lead to extremely high area and power overhead that in some cases, such as triple modular redundancy, can reach up to 200%. Consequently, this is not an affordable approach to provide defect tolerance to mainstream cost-sensitive microprocessor designs. To this end, this thesis suggests that the same degree of defect tolerance can be provided at a much lower cost by periodically checking the integrity of the underlying hardware rather than continuously monitoring the execution for errors.

To demonstrate the feasibility and effectiveness of the periodic hardware checking defect-tolerance paradigm, this thesis proposed the BulletProof approach. The BulletProof approach augments the processor with a microarchitectural checkpointing and recovery mechanism that provides a substrate for speculative computation epochs. After each speculative computation epoch, distributed component-specific on-chip checkers run BIST-like tests to verify the integrity of the underlying hardware components. Additionally, a double-sampling flip-flop design is used to detect transient fault logic glitches that can corrupt the pipeline state. If, at the end of an epoch, the hardware is fault-free, the epoch computation is allowed to retire to non-speculative state. In the event that a fault is exposed, the program state is rolled back to the last known good program state at the beginning of the last epoch.

To evaluate the effectiveness of the BulletProof approach, we developed a physical-level prototype of a 4-wide VLIW processor augmented with on-chip component-specific hardware checkers. Based on the prototype implementation, we found that the BulletProof technique can provide about 95% defect coverage to the design, for an area overhead of 14%, and a runtime performance overhead of less than 1%. Although the runtime performance overhead of BulletProof is negligible, and its area overhead is extremely low

compared to that of traditional defect-tolerance techniques, the area overhead is still high for its adoption in cost-sensitive designs. Furthermore, the defect coverage is lower than the almost 100% coverage provided by traditional defect-tolerance techniques. However, the negligible runtime performance overhead of BulletProof allowed us to trade-off performance overhead with higher defect coverage and lower area overhead in order to reach our goal of very high defect coverage for very low area overhead. To enable that trade-off, this thesis proposed another novel approach by moving the defect detection and diagnosis from the on-chip hardware checkers to software routines that are able to test the underlying hardware for defects.

We called this new novel software-based hardware testing approach the Access-Control Extension (ACE) Framework. The ACE framework allows special ISA instructions to access and control virtually any part of the processor's internal state. Based on this framework, special firmware periodically suspends the processor's execution and performs high-quality testing of the underlying hardware to detect defects by exercising the hardware with pre-generated high-quality ATPG test patterns. The use of these software testing routines eliminates the need for the on-chip hardware checkers used in the BulletProof approach. However, the other techniques employed by the BulletProof approach, such as microarchitectural checkpointing and recovery that enables the periodic hardware checking, the online hardware reconfiguration techniques used for hardware repair, and the double-latching flip-flops to provide transient-fault tolerance, are still used in combination with the hardware testing capabilities of the ACE framework to provide a comprehensive defect-tolerance solution.

The experimental evaluation of the ACE framework was done on a commercial multi-core processor design that is based on Sun's Niagara. Based on our experimental evaluation, we found that the ACE testing is capable of performing high-quality hardware testing for 99.22% of the chip area. We also found that, based on a detailed RTL implementation of the ACE framework, augmenting the Sun Niagara processor with the ACE framework results in a 5.8% increase in chip area and a 4% increase in power consumption. We also found that the runtime performance overhead of the ACE framework is around 5% when the underlying hardware is tested for stuck-at faults, the industry standard fault model used for manufacturing testing. These experimental results demonstrate that by combining the periodic hardware checking approach of BulletProof with the software-based hardware checking of the ACE framework, we can develop defect-tolerance solutions that can provide very high defect coverage that is close to 100%, for a very low area cost of around 6%, and a low runtime performance slowdown of 5%. This makes the case for this thesis, that is, it is indeed possible to develop defect-tolerance solutions for microprocessor

designs that can provide the same reliability guarantees as the traditional defect-tolerance solutions, but at a much lower cost.

Furthermore, we believe that the ACE framework capability of providing hardware accessibility and controllability to the software, that we originally developed for running software routines that test the hardware, is a more generic feature that can be found useful in many other important applications. We believe that this extensive use of the ACE framework adds value to the mechanism and it can ease its possible adoption in future generation microprocessors. To demonstrate the extensive use of the ACE framework to other applications, this thesis described how the ACE framework hardware resources can be extended to three other applications: i) for the online detection of design bugs, ii) as a post-silicon debugging tool, and iii) for improving the manufacturing testing process.

Finally, in order to quantify the microprocessor reliability requirements that need to be addressed by defect-tolerance techniques like the BulletProof and the ACE framework, we first need to assess the severity of the reliability threats to a microprocessor design using a resiliency analysis tool. To this end, this thesis concludes with the development of CrashTest, a novel FPGA-based framework for the accurate resiliency analysis of modern microprocessor designs. The CrashTest is different from previously proposed hardware resiliency analysis tools because it can automatically orchestrate a fault injection and analysis campaign on the gate-level netlist of the design, while employing FPGA-based accelerated hardware emulation to enable the analysis of complex full-system designs which can boot an operating system and run applications. Furthermore, the CrashTest framework supports an extended collection of fault models ranging from transient faults to silicon defects, and it can easily be upgraded with new fault models. We found that for the resiliency evaluation of the LEON3 system-on-chip, the use of a prototype implementation of CrashTest that was developed on a commercial FPGA provided a six orders of magnitude speedup compared to an equivalent software-based hardware resiliency analysis simulator.

7.2 Thesis Conclusions

This thesis provided a new thinking in the design of microprocessor defect-tolerance solutions through the techniques described in Chapters III-V and proposed a novel approach for evaluating the resiliency of microprocessor designs in Chapter VI. Based on the exploration of these novel techniques, this thesis draws the following conclusions:

- **The BulletProof Approach - Periodic Hardware Checking:** The BulletProof approach, presented in Chapter III, is notably different from traditional approaches to

fault tolerance. Specifically, it shifts the traditional defect-tolerance paradigm from continuous checking for execution errors to periodic online hardware checking. This approach is markedly different from the traditional defect-tolerance approach and is achieved by using a microarchitectural checkpointing mechanism that creates speculative epochs of computation and on-chip hardware checkers.

For the evaluation of BulletProof, we implemented a physical prototype of the BulletProof mechanism, based on a 4-wide VLIW processor, and we found that the area overhead of the BulletProof mechanism is quite modest, providing transient and hard silicon fault protection with only a 14% increase in total area. This is a remarkable improvement over traditional redundancy-based techniques, such as triple-modular redundancy, which incurs overheads starting at 200%. Additionally, it was demonstrated through gate-level fault injection studies that fault-detection coverage is high: 95% of all hard silicon defects and 99% of all transient faults are covered. However, although BulletProof has a significant improvement in terms of area overhead over the traditional defect-tolerance techniques, its area overhead of 14% is still high for its adoption in mainstream cost-sensitive microprocessors, and its defect coverage of 95% is still a drawback against the almost 100% coverage of traditional techniques.

- **The ACE Framework - Software-Based Testing:** To lower the cost of the BulletProof mechanism and provide more flexible hardware checking strategies with higher defect coverage, the Access-Control Extension (ACE) Framework, presented in Chapter IV, shifted the silicon defect detection and diagnosis process from on-chip hardware checkers to software. This new approach, enabled the trade-off of runtime performance overhead with lower area overhead for testing and higher defect coverage.

  We experimentally evaluated the ACE framework on a commercial multicore processor design based on Sun's Niagara and we found that ACE testing is capable of performing high-quality hardware testing for 99.22% of the chip area. We also found that, based on a detailed RTL implementation, the ACE framework requires a 5.8% increase in Sun Niagara's chip area and a 4% increase in its power consumption envelope. We also measured the runtime performance of ACE testing and we found it to be around 5% for test patternd generated using the stuck-at fault model, the industry standard fault model used for manufacturing testing.

  Based on the experimental evaluation of the ACE framework, we conclude that: 1) it can effectively remove the need for on-chip hardware checkers used in the Bullet-

Proof approach and move this functionality to software, 2) it has ample flexibility to be modified/upgraded in the field because it is not hardwired in the design, 3) it can be uniformly applied to any microprocessor module with low design complexity because it does not require module-specific customizations, and 4) it can provide wide coverage across the whole chip, including non-core modules.

Overall, based on our experimental evaluation, we conclude that with the combination of BulletProof-based periodic hardware testing with the ACE software-based hardware checking routines, this thesis makes a strong case that it is possible to develop online defect-tolerance solutions for microprocessor designs that provide the same reliability guarantees as traditional techniques, but at a much lower cost.

- **ACE Framework Extensions - Adding Value to Resiliency Mechanisms:** Chapter V demonstrated that the ACE framework can be extended to other important applications to amortize its cost and ease its adoption in future generation microprocessor designs.

  The first application considered as an ACE framework hardware extension was online design bug detection. In that context, we provided a rigorous analysis of processor design bugs in the RTL code of a commercial microprocessor. Based on the insights obtained from our rigorous design bug analysis, we proposed a novel distributed online bug detection mechanism based on the ACE framework. We also described how the ACE framework can be extended to improve the quality and reduce the of cost post-silicon debugging and manufacturing testing.

  The cost of the extended ACE framework was evaluated on a detailed RTL prototype implementation and we found that the total silicon area overhead incurred is 15% of the whole OpenSPARC T1 chip, while the power consumption overhead is only 6.8%.

  Based on these numbers, we conclude that the ACE framework is a general framework that can be used for multiple purposes to enhance the reliability and to reduce the design/testing cost of modern microprocessors and that it can provide additional value for its cost, something that would make its possible adoption by future generation microprocessors easier.

- **The CrashTest Framework - FPGA-Accelerated Resiliency Analysis:** In Chapter VI, we presented CrashTest, a novel FPGA-based framework for the accurate resiliency analysis of modern microprocessor designs. The CrashTest framework can automatically orchestrate a fault injection and analysis campaign on the gate-level

netlist of the design, while employing FPGA-based accelerated hardware emulation to enable the analysis of complex full-system designs which can boot an operating system and run applications.

We evaluated the the CrashTest framework by performing gate-level fault injection campaigns on the netlist of a LEON3 system-on-chip while booting an unmodified version of Linux 2.6 operating system using a commercial FPGA-based platform. From these experiments, we found that the use of hardware emulation, when compared to an equivalent software-based hardware resiliency analysis simulator, it can accelerate the fault simulation and analysis process by six orders of magnitude. Based on these experimental results, we conclude that the proposed CrashTest framework can provide both a high-performance and a high-fidelity hardware resiliency analysis tool for complex modern microprocessor designs.

7.3 Future Work

The work presented in this thesis also opens the door to several future research directions. The microprocessor defect-tolerance solutions presented in this thesis rely on microarchitectural resource redundancy that is present in most modern multicore processors. In particular, the proposed approach for repairing the underlying hardware is by disabling any defective parts and continue operation with the remaining resources in a performance and/or capability degraded mode. However, the extend to which this approach is effective depends on the amount and nature of the microarchitectural resource redundancy that is present in the processor. For example, if the microprocessor is comprised by thousands of simple and very small processing elements, as is proposed in tile architectures [128], the performance degradation of losing some of those processing elements to silicon defects could be insignificant or even unnoticed. On the other spectrum of the design space, if the microprocessor is comprised by very few monolithic cores interconnected with unique architectural components (*e.g.*, I/O buses and memory controllers), the lost of even a single component to silicon defects can seriously impair the microprocessor's performance and functionality. Although tile-style architectures provide an attractive solution to this problem, they have their own drawbacks and as of today no commercial microprocessor has adopted this style of architecture. As a future research direction, it would be interesting to investigate microprocessor design techniques that would make the hardware resource reconfiguration more effective and tolerant to silicon defects and explore the trade-off across the spectrum of the architecture design space.

Furthermore, another interesting research direction it would be to investigate how defect tolerance could be moved from a hardware responsibility to a software feature. Today,

the correct execution of software relies on the assumption that the underlying hardware are defect-free and are functionality correct virtually 100% of the time. By breaking this assumption, we essentially move the correctness responsibility from the hardware to the software. It would be interesting to investigate if it is possible to develop resilient algorithms that can guarantee software correctness in the presence of an unreliable hardware computing substrate. Such software solutions, could be a promising alternative solution for making possible the transition into future highly unreliable silicon process technologies.

Altogether, the defect-tolerance solutions presented in this thesis provide a cost-effective framework that enables the development of reliable microprocessors with unreliable silicon components. Furthermore, it was demonstrated that the hardware resources of the proposed defect-tolerance solutions can be utilized by other important applications to amortize their cost and ease their adoption by future generation microprocessor designs. We hope, that the contributions made by the work presented in this thesis advance the research area of microprocessor defect-tolerance design and that the techniques proposed in this thesis will find applicability in future commercial microprocessor designs.

BIBLIOGRAPHY

[1] Advanced Micro Devices. Revision Guide for AMD Athlon 64 and AMD Opteron Processors, Pub. No. 25759 Rev. 3.75, Feb. 2008.

[2] A. Agarwal, B.-H. Lim, D. A. Kranz, and J. Kubiatowicz. April: A processor architecture for multiprocessing. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 1990.

[3] N. Aggarwal, P. Ranganathan, N. P. Jouppi, and J. E. Smith. Configurable isolation: building high availability systems with commodity multi-core processors. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 470–481, June 2007.

[4] T. Austin, V. Bertacco, S. Mahlke, and Y. Cao. Reliable systems on unreliable fabrics. *IEEE Design & Test of Computers*, 25(4):322–332, 2008.

[5] T. Austin, D. Blaauw, T. Mudge, and K. Flautner. Making typical silicon matter with Razor. *IEEE Computer*, 37(3):57–65, 2004.

[6] T. M. Austin. DIVA: A reliable substrate for deep submicron microarchitecture design. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 196–207, 1999.

[7] A. Avžienis. Arithmetic error codes: Cost and effectiveness studies for application in digital system design. *IEEE Transactions on Computers (IEEE TC)*, C-20(II):1322–1331, 1971.

[8] A. Avžienis and Y. He. Microprocessor entomology: A taxonomy of design faults in COTS microprocessors. In *DCCA-99*, pages 3–24, 1999.

[9] F. Bacchini, R. Damiano, B. Bentley, K. Baty, K. Normoyle, M. Ishii, and E. Yogev. Verification: what works and what doesn't. In *Proceedings of the Design Automation Conference (DAC)*, 2004.

[10] J. E. Bartlett, J. W. Kotrlik, and C. C. Higgins. Organizational research: Determining appropriate sample size in survey research. *Information Technology, Learning, and Performance Journal*, 19(1):43–50, 2001.

[11] K. Batcher and C. Papachristiou. Instruction randomization self test for processor cores. In *IEEE VLSI Test Symposium (VTS)*, 1999.

[12] J. M. Berger. A note on error detection codes for asymmetric channels. *Information and Control*, 4(1):68–73, 1961.

[13] P. Bernardi, L. Bolzani, M. Rebaudengo, M. Reorda, F. Vargas, and M. Violante. A new hybrid fault detection technique for systems-on-a-chip. *IEEE Transactions on Computers*, 55(2):185–198, February 2006.

[14] J. A. Blome, S. Feng, S. Gupta, and S. A. Mahlke. Self-calibrating online wearout

detection. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 109–122, 2007.

[15] S. Borkar, T. Karnik, and V. De. Design and reliability challenges in nanometer technologies. In *Proceedings of the Design Automation Conference (DAC)*, 2004.

[16] B. Bose and D. J. Lin. Systematic unidirectional error-detecting codes. *IEEE Transactions on Computers (IEEE TC)*, 34(11):1026–1032, 1985.

[17] P. Bose. Designing reliable systems with unreliable components. *IEEE Micro*, 26(5):5–6, 2006.

[18] B. Bottoms. The third millennium's test dilemma. *IEEE Design & Test of Computers*, 15(4):7–11, 1998.

[19] F. A. Bower, P. G. Shealy, S. Ozev, and D. J. Sorin. Tolerating hard faults in microprocessor array structures. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, June 2004.

[20] F. A. Bower, D. J. Sorin, and S. Ozev. A mechanism for online diagnosis of hard faults in microprocessors. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, Nov. 2005.

[21] D. Brahme and J. A. Abraham. Functional testing of microprocessors. *IEEE Transactions on Computers*, C-33:475–485, 1984.

[22] Brian T. Murray and John P. Hayes. Testing ICs: Getting to the Core of the Problem. *IEEE Computer*, 29(11):32–38, 1996.

[23] M. L. Bushnell and V. D. Agrawal. *Essentials of Electronic Testing for Digital, Memory and Mixed-Signal VLSI Circuits*. Kluwer Academic Publishers, Boston, 2000.

[24] K.-H. Chang, I. L. Markov, and V. Bertacco. Automating post-silicon debugging and repair. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, Nov. 2007.

[25] K. Chen, S. Malik, and P. Patra. Runtime validation of transactional memory systems. In *Proceeding of the International Symposium on Quality Electronic Design (ISQED)*, pages 750–756, 2008.

[26] L. Chen and S. Dey. Software-based self-testing methodology for processor cores. *IEEE TCAD*, 20(3):369–380, 2001.

[27] P. Civera, L. Macchiarulo, M. Rebaudengo, M. S. Reorda, and M. Violante. FPGA-based fault injection techniques for fast evaluation of fault tolerance in VLSI circuits. *Lecture Notes in Computer Science*, 2147, 2001.

[28] K. Constantinides, O. Mutlu, T. Austin, and V. Bertacco. Software-based online detection of hardware defects: Mechanisms, architectural support, and evaluation. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2007.

[29] W. J. Dally, L. R. Dennison, D. Harris, K. Kan, and T. Xanthopoulos. The reliable router: A reliable and high-performance communication substrate for parallel computers. In *Parallel Computer Routing and Communication, First International Workshop, PCRCW*, pages 241–255, May 1994.

[30] A. DeOrio, A. Bauserman, and V. Bertacco. Chico: An on-chip hardware checker for pipeline control logic. In *Proceedings of the International Workshop on Microprocessor Test and Verification (MTV)*, 2007.

[31] E. Dupont, D. Chardonnereau, R. Keulen, and D. Alexandrescu. Design of a resilient processor. In *Workshop on System Effects of Logic Soft Errors (SELSE)*, 2005.

[32] N. Durrant and R. Blish. Semiconductor device reliability failure models. Available at *http://www.sematech.org/*, 2000.

[33] D. Ernst, N. S. Kim, S. Das, S. Pant, R. R. Rao, T. Pham, C. H. Ziesler, D. Blaauw, T. M. Austin, K. Flautner, and T. N. Mudge. Razor: A low-power pipeline based on circuit-level timing speculation. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 7–18, 2003.

[34] D. Fick, A. DeOrio, G. Chen, V. Bertacco, D. Sylvester, and D. Blaauw. A highly resilient routing algorithm for fault-tolerant nocs. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE)*, 2009.

[35] M. J. Flynn and P. Hung. Microprocessor design issues: Thoughts on the road ahead. *IEEE Micro*, 25(3), 2005.

[36] Freescale Semiconductor. MPC7457 Chip Errata, Rev. 10, Nov. 2004.

[37] J. Gaisler. A portable and fault-tolerant microprocessor based on the SPARC V8 architecture. In *Proceedings of International Conference on Dependable Systems and Networks (DSN)*, pages 409–415, 2002.

[38] A. Gluska. Coverage-oriented verification of Banias. In *Proceedings of the Design Automation Conference (DAC)*, June 2003.

[39] R. Goering. Post-silicon debugging worth a second look. In *EETimes*, Feb. 5 2007.

[40] R. Guo, S. Mitra, E. Amyeen, J. Lee, S. Sivaraj, and S. Venkataraman. Evaluation of test metrics: Stuck-at, bridge coverage estimate and gate exhaustive. In *IEEE VLSI Test Symposium (VTS)*, pages 66–71, 2006.

[41] P. Gupta and A. B. Kahng. Manufacturing-aware physical design. In *Proceedings of International Conference on Computer-Aided Design (ICCAD)*, pages 681–687, 2003.

[42] S. Gupta, S. Feng, A. Ansari, J. Blome, and S. Mahlke. The StageNet fabric for constructing resilient multicore systems. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2008.

[43] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *IEEE Annual Workshop on Workload Characteristics*, pages 3–14, 2001.

[44] J. Hennessy and D. Patterson. *Computer Architecture a Quantitative Approach*. Morgan Kaufmann Publishers, Inc., 1996.

[45] G. Hetherington, T. Fryars, N. Tamarapalli, M. Kassab, A. Hassan, and J. Rajski. Logic BIST for large industrial designs: real issues and case studies. In *Proceedings of the International Test Conference (ITC)*, pages 358–367, September 1999.

[46] Hewlett-Packard Company. NetPerf: A network performance benchmark. 1995.

[47] M. D. Hill and A. J. Smith. Evaluating associativity in CPU caches. *IEEE Transactions on Computers (IEEE TC)*, 38(12):1612–1630, 1989.

[48] H. Hirata, K. Kimura, S. Nagamine, Y. Mochizuki, A. Nishimura, Y. Nakase, and T. Nishizawa. An elementary processor architecture with simultaneous instruction issuing from multiple threads. In *Proceedings of the International Symposium on*

Computer Architecture (ISCA), 1992.

[49] H. Holzapfel and P. Levin. Advanced post-silicon verification and debug. *EDA Tech Forum*, 3(3), September 2006.

[50] C. K. Hu and R. Rosenberg. Scaling effect on electromigration in on-chip Cu wiring. *International Electron Devices Meeting*, 1999.

[51] Intel Corporation. Intel Xeon Processor - Specification Update, Doc. No. 249678-056, Dec. 2006.

[52] Intel Corporation. Intel Core 2 Extreme Processor X6800 and Intel Core 2 Duo Desktop Processor E6000 and E4000 - Specification Update, Doc. No. 313279-024, Feb. 2008.

[53] Intel Corporation. Intel Core Duo Processor and Intel Core Solo Processor on 65nm Process - Specification Update, Doc. No. 309222-016, Feb. 2008.

[54] Intel Corporation. Intel Pentium M Processor - Specification Update, Doc. No. 252665-033, Jan. 2008.

[55] Intel Corporation. Intel Pentium 4 Processor - Specification Update, Doc. No. 249199-069, May 2007.

[56] Intel Corporation. Intel Pentium 4 Processor on 90nm Process - Specification Update, Doc. No. 302352-031, Sept. 2006.

[57] A. M. Ionescu, M. J. Declercq, S. Mahapatra, K. Banerjee, and J. Gautier. Few electron devices: towards hybrid CMOS-SET integrated circuits. In *Proceedings of the Design Automation Conference (DAC)*, pages 88–93, 2002.

[58] B. Janssens and W. K. Fuchs. The performance of cache-based error recovery in multiprocessors. *IEEE Transactions on Parallel Distributed Systems*, 5(10):1033–1043, 1994.

[59] Joint Electron Device Engineering Council. Failure mechanisms and models for semiconductor devices. *JEDEC Publication JEP122-A*, 2002.

[60] D. Josephson. The good, the bad, and the ugly of silicon debug. In *Proceedings of the Design Automation Conference (DAC)*, pages 3–6, 2006.

[61] D. Josephson and B. Gottlieb. The crazy mixed up world of silicon debug. In *Proceedings of the Custom Integrated Circuits Conference*, pages 665–670, October 2004.

[62] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham. FERRARI: A flexible software-based fault and error injection system. *IEEE Transactions on Computers (IEEE TC)*, 44(2):248–260, 1995.

[63] H. Klug. Microprocessor testing by instruction sequences derived from random patterns. In *Proceedings of the International Test Conference (ITC)*, 1988.

[64] C. Kong. A hardware overview of the NonStop Himalaya (K10000). *Tandem Systems Overview*, 10(1):4–11, Jan. 1994.

[65] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way multithreaded SPARC processor. *IEEE Micro*, 25(2):21–29, 2005.

[66] N. Kranitis, A. Paschalis, D. Gizopoulos, and Y. Zorian. Instruction-based self-test of processor cores. In *IEEE VLSI Test Symposium (VTS)*, 2002.

[67] R. Kuppuswamy, P. DesRosier, D. Feltham, R. Sheikh, and P. Thadikaran. Full hold-scan systems in microprocessors: Cost/benefit analysis. *Intel Technology*

Journal (ITJ), 8(1):63–72, Feb. 2004.

[68] P. Lala. *Self-Checking and Fault-Tolerant Digital Design, 1st Edition*. Morgan Kaufmann Publisher, 2001.

[69] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: A tool for evaluating and synthesizing multimedia and communicatons systems. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 1997.

[70] J. lee and J. H. Patel. An instruction sequence assembling methodology for testing microprocessors. In *Proceedings of International Test Conference (ITC)*, 1992.

[71] Y.-H. Lee, N. Mielke, M. Agostinelli, S. Gupta, R. Lu, and W. McMahon. Prediction of logic product failure due to thin-gate oxide breakdown. In *Proceedings of the International Reliability Physics Symposium*, pages 18–28, March 2006.

[72] A. S. Leon, K. W. Tam, J. L. Shin, D. Weisner, and F. Schumacher. A power-efficient high-throughput 32-thread SPARC processor. *IEEE Journal of Solid-State Circuits*, 42(1):7–16, January 2006.

[73] M.-L. Li, P. Ramachandran, S. K. Sahoo, S. V. Adve, V. S. Adve, and Y. Zhou. Understanding the propagation of hard errors to software and implications for resilient system design. In *Proceedings of the International Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008.

[74] Y. Li, S. Makar, and S. Mitra. CASP: Concurrent autonomous chip self-test using stored test patterns. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE)*, 2008.

[75] X. Lin, R. Press, J. Rajski, P. Reuter, T. Rinderknecht, B. Swanson, and N. Tamarapalli. High-frequency, at-speed scan testing. *IEEE Design and Test of Computers*, 20(5):17–25, Sep-Oct 2003.

[76] C. López-Ongil, M. García-Valderas, M. Portela-García, and L. Entrena-Arrontes. An autonomous FPGA-based emulation system for fast fault tolerant evaluation. In *FPL*, pages 397–402, 2005.

[77] C.-K. Luk, R. S. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood". Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the Programming Language Design and Implementation Conference (PLDI)*, pages 190–200, 2005.

[78] M. Magee. Intel's hidden Xeon, Pentium 4 bugs. http://www.theinquirer.net, Aug. 2002.

[79] G. Magklis, M. L. Scott, G. Semeraro, D. H. Albonesi, and S. Dropsho. Profile-based dynamic voltage and frequency scaling for a multiple clock domain microprocessor. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, June 2003.

[80] J. F. Martinez, J. Renau, M. C. Huang, M. Prvulovic, and J. Torrellas. Cherry: Checkpointed early resource recycling in out-of-order microprocessors. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 3–14, 2002.

[81] D. G. Mavis and P. H. Eaton. Soft error rate mitigation techniques for modern microcircuits. In *Proceedings of 40th Annual Reliability Physics Symposium*, pages 216–225, 2002.

[82] E. J. McCluskey and C.-W. Tseng. Stuck-fault tests vs. actual defects. In *Proceedings of the International Test Conference (ITC)*, pages 336–343, October 2000.

[83] M. Mehrara, M. Attariyan, S. Shyam, K. Constantinides, V. Bertacco, and T. M. Austin. Low-cost protection for SER upsets and silicon defects. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE)*, pages 1146–1151, 2007.

[84] Mei-Chen, T. K. Tsai, and R. K. Iyer. Fault injection techniques and tools. *IEEE Computer*, 30(4):75–82, Apr. 1997.

[85] A. Meixner, M. E. Bauer, and D. J. Sorin. Argus: Low-cost, comprehensive error detection in simple cores. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2007.

[86] A. Meixner and D. J. Sorin. Error detection using dynamic dataflow verification. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 104–118, 2007.

[87] M. Meterelliyoz, H. Mahmoodi, and K. Roy. A leakage control system for thermal stability during burn-in test. *Proceedings of International Test Conference (ITC)*, Nov. 2005.

[88] N. Miskov-Zivanov and D. Marculescu. MARS-C: modeling and reduction of soft errors in combinational circuits. In *Proceedings of the Design Automation Conference (DAC)*, pages 767–772, 2006.

[89] S. Mitra and E. J.McCluskey. Which concurrent detection scheme to choose? In *Proceedings of the International Test Conference (ITC)*, pages 985–994, 2000.

[90] S. Mitra, N. Seifert, M. Zhang, Q. Shi, and K. S. Kim. Robust system design with built-in soft-error resilience. *IEEE Computer*, 38(2):43–52, Feb. 2005.

[91] S. Mitra, M. Zhang, N. Seifert, B. Gill, S. Waqas, and K. S. Kim. Combinational logic soft error correction. In *Proceedings of International Test Conference (ITC)*, November 2006.

[92] G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, 1965.

[93] S. Mukherjee, J. Emer, and S. Reinhardt. The soft error problem: An architectural perspective. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2005.

[94] A. Murakami, S. Kajihara, T. Sasao, I. Pomeranz, and S. M. Reddy. Selection of potentially testable path delay faults for test generation. In *Proceedings of the International Test Conference (ITC)*, pages 376–384, Oct. 2000.

[95] J. Nakano, P. Montesinos, K. Gharachorloo, and J. Torrellas. ReViveI/O: Efficient handling of I/O in highly-available rollback-recovery servers. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2006.

[96] S. Narayanasamy, B. Carneal, and B. Calder. Patching processor design errors. In *Proceedings of the International Conference on Computer Design (ICCD)*, October 2006.

[97] M. Nicolaidis. Time redundancy based soft-error tolerance to rescue nanometer technologies. In *IEEE VLSI Test Symposium (VTS)*, pages 86–94, 1999.

[98] E. B. Nightingale, P. M. Chen, and J. Flinn. Speculative execution in a distributed file system. *ACM Transactions on Computer Systems*, 24(4):361–392, 2006.

[99] P. Parvathala, K. Maneparambil, and W. Lindsay. FRITS - A microprocessor functional BIST method. In *Proceedings of the International Test Conference (ITC)*, pages 590–598, Oct. 2002.

[100] L.-S. Peh. *Flow Control and Micro-Architectural Mechanisms for Extending the Performance of Interconnection Networks*. PhD thesis, Stanford University, 2001.

[101] M. Prvulovic, Z. Zhang, and J. Torrellas. ReVive: Cost-effective architectural support for rollback recovery in shared-memory multiprocessors. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 111–122, May 25–29 2002.

[102] B. R. Quinton and S. J. E. Wilton. Post-silicon debug using programmable logic cores. In *Proceedings of the International Conference on Field-Programmable Technology (FPT)*, pages 241–248, December 2005.

[103] J. M. Rabaey. *Digital integrated circuits: a design perspective*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[104] R. Rao, A. Srivastava, D. Blaauw, and D. Sylvester. Statistical estimation of leakage current considering inter- and intra-die process variation. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*, pages 84–89, 2003.

[105] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August. SWIFT: Software implemented fault tolerance. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, pages 243–254, 2005.

[106] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Privulovic, L. Ceze, S. Sarangi, P. Sack, K. Stauss, and P. Montesinos. SESC Simulator. http://sesc.sourceforge.net, 2002.

[107] G. P. Saggese, A. Vetteth, Z. Kalbarczyk, and R. K. Iyer. Microprocessor sensitivity to failures: Control vs execution and combinational vs sequential logic. In *Proceedings of International Conference on Dependable Systems and Networks (DSN)*, pages 760–769, 2005.

[108] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore. Exploiting ILP, DLP, and TLP using polymorphism in the TRIPS architecture. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 422–433, 2003.

[109] S. Sarangi, S. Narayanasamy, B. Carneal, A. Tiwari, B. Calder, and J. Torrellas. Patching processor design errors with programmable hardware. *IEEE Micro*, 2007.

[110] S. R. Sarangi, A. Tiwari, and J. Torrellas. Phoenix: Detecting and recovering from permanent processor design bugs with programmable hardware. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 26–37, 2006.

[111] E. Schuchman and T. N. Vijaykumar. Rescue: A microarchitecture for testability and defect tolerance. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 160–171, 2005.

[112] M. J. Serrano, W. Yamamoto, R. C. Wood, and M. Nemirovsky. A model for performance estimation in a multistreamed, superscalar processor. In *Proceedings of the International Conference on Modeling Techniques and Tools for Computer Per-*

formance Evaluation, 1994.

[113] P. P. Shirvani, N. Saxena, and E. J. Mccluskey. Software-implemented EDAC protection against SEUs. *IEEE Transactions on Reliability*, 49:273–284, 2000.

[114] P. Shivakumar, S. W. Keckler, C. R. Moore, and D. Burger. Exploiting microarchitectural redundancy for defect tolerance. In *Proceedings of the International Conference on Computer Design (ICCD)*, 2003.

[115] M. Shulz. The end of the road for silicon. *Nature Magazine*, June 1999.

[116] S. Shyam, K. Constantinides, S. Phadke, V. Bertacco, and T. Austin. Ultra low-cost defect protection for microprocessor pipelines. In *Proceedings of the International Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 73–82, 2006.

[117] D. P. Siewiorek and R. S. Swarz. Reliable computer systems: Design and evaluation, 3rd edition. *AK Peters, Ltd*, 1998.

[118] J. Smolens, B. Gold, J. Kim, B. Falsafi, J. Hoe, and A. Nowatzyk. Fingerprinting: Bounding the soft-error detection latency and bandwidth. In *Proceedings of the International Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2004.

[119] J. C. Smolens, B. T. Gold, J. C. Hoe, B. Falsafi, and K. Mai. Detecting emerging wearout faults. In *Workshop on System Effects of Logic Soft Errors (SELSE)*, 2007.

[120] D. J. Sorin, M. M. K. Martin, M. D. Hill, and D. A. Wood. SafetyNet: Improving the availability of shared memory multiprocessors with global checkpoint/recovery. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 123–136, May 25–29 2002.

[121] L. Spainhower and T. A. Gregg. G4: A fault-tolerant CMOS mainframe. In *Proceedings of the International Symposium on Fault-Tolerant Computing (FTCS)*, 1998.

[122] J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers. The case for lifetime reliability-aware microprocessors. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 276–287, 2004.

[123] J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers. The impact of technology scaling on lifetime reliability. In *Proceedings of International Conference on Dependable Systems and Networks (DSN)*, pages 177–186, 2004.

[124] J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers. Exploiting structural duplication for lifetime reliability enhancement. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2005.

[125] J. H. Stathis. Reliability limits for the gate insulator in CMOS technology. *IBM Journal of Research and Development*, 46(2/3):265–286, 2002.

[126] Sun Microsystems Inc. OpenSPARC T1. http://opensparc-t1.sunsource.net/.

[127] Sun Microsystems Inc. OpenSPARC T1 Microarchitecture Specification. August 2006.

[128] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin. Wavescalar. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2003.

[129] D. Sylvester, D. Blaauw, and E. Karl. ElastIC: An adaptive self-healing architecture for unpredictable silicon. *IEEE Design & Test of Computers*, 23(6):484–490, 2006.

[130] Synopsys. TetraMAX ATPG User Guide, Version 2002.05. http://www.synopsys.com, 2002.

[131] Synopsys. Design Compiler User Guide, Version 2002.05, Jun. 2002.

[132] D. Tarjan, S. Thoziyoor, and N. P. Jouppi. Cacti 4.0. Technical report hpl-2006-86, Hewlett-Packard, 2006.

[133] M. H. Tehranipour, S. Fakhraie, Z. Navabi, and M. Movahedin. A low-cost at-speed bist architecture for embedded processor and sram cores. *Journal of Electronic Testing: Theory and Applications*, 20(2):155–168, 2004.

[134] R. Theodorescu, J. Nakano, and J. Torrellas. SWICH: A prototype for efficient cache-level checkpointing and rollback. *IEEE Micro*, 2006.

[135] Trimaran. An infrastructure for research in ILP. http://www.trimaran.org, 2000.

[136] D. Tullsen, S. Eggers, and H. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, June 1995.

[137] D. P. Vallett. Future challenges in IC testing and fault isolation. *Proceedings of the Annual Meeting of the IEEE Lasers and Electro-Optics Society (LEOS)*, 2:539–540, October 2003.

[138] S. B. K. Vrudhula, D. Blaauw, and S. Sirichotiyakul. Estimation of the likelihood of capacitive coupling noise. In *Proceedings of the Design Automation Conference (DAC)*, 2002.

[139] I. Wagner and V. Bertacco. Engineering trust with semantic guardians. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE)*, pages 743–748, 2007.

[140] I. Wagner, V. Bertacco, and T. Austin. Shielding against design flaws with field repairable control logic. In *Proceedings of the Design Automation Conference (DAC)*, 2006.

[141] I. Wagner, V. Bertacco, and T. Austin. Using field-repairable control logic to correct design errors in microprocessors. *IEEE Transactions on computer-aided design of integrated circuits and systems*, 27(2):380–393, Feb 2008.

[142] N. J. Wang, J. Quek, T. M. Rafacz, , and S. J. Patel. Characterizing the effects of transient faults on a high-performance processor pipeline. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pages 61–70, 2004.

[143] C. Weaver and T. Austin. A fault tolerant approach to microprocessor design. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pages 411–420, 2001.

[144] 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 International Symposium on Computer Architecture (ISCA)*, 2004.

[145] A. Wolfe. For Intel, it's a case of FPU all over again. In *EE Times*, May 1997.

[146] T. J. Wood. The test and debug features of the AMD-K7 microprocessor. In *Proceedings of the International Test Conference (ITC)*, pages 130–136, 1999.

[147] E. Y. Wu and J. Suñé. Interplay of voltage and temperature acceleration of oxide breakdown for ultra-thin gate dioxides. *Solid-state Electronics Journal*, 2002.

[148] Xilinx. Xilinx University Program: Xilinx XUP Virtex-II Pro Development System. http://www.xilinx.com/univ/xupv2p.html, 2005.

[149] W. M. Yee, M. Paniccia, T. Eiles, and V. Rao. Laser voltage probe (LVP): A novel optical probing technology for flip-chip packaged microprocessors. In *Proceedings of the International Symposium of Physical and Failure Analysis of Integrated Circuits (IPFA)*, pages 15–20, 1999.

[150] B. Zhang, W.-S. Wang, and M. Orshansky. FASER: Fast analysis of soft error susceptibility for cell-based designs. In *Proceeding of the International Symposium on Quality Electronic Design (ISQED)*, pages 755–760, 2006.

[151] J. F. Ziegler. Terrestrial cosmic rays. *IBM Journal of Research and Development*, 40(1), 1996.