

© 2009 Man-Lap Li

SWAT: DESIGNING RESILIENT HARDWARE BY TREATING SOFTWARE ANOMALIES

BY

MAN-LAP LI

B.S., University of California, Berkeley 2001

M.S., University of Illinois at Urbana-Champaign, 2005

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2009

Urbana, Illinois

Doctoral Committee:

Professor Sarita V. Adve, Chair

Associate Professor Vikram Adve

Professor Marc Snir

Assistant Professor Craig Zilles

Professor Yuanyuan Zhou, University of California, San Diego

Dr. Pradip Bose, IBM Research

Abstract

With continued CMOS scaling, future shipped hardware will be increasingly vulnerable to faults and inevitably fail in the field for a variety of reasons such as aging or wear-out, radiation, infant mortality due to inadequate burn-in, design defects, manufacturing defects, and so on. Further, this reliability threat is expected to pervade even the mainstream computing market, making traditional solutions that involve redundancy in space, time, and/or information too expensive to be broadly deployable. Hence, there is a need for effective in-field reliability solutions that incur low overheads in area, power, and performance and handle multiple sources of errors.

This dissertation proposes a low-cost comprehensive reliability solution that detects, diagnoses, and recovers from in-field errors. Our design is based on the following two key observations. (1) Hardware reliability solutions only need to handle device faults that manifest in software. (2) Despite the growing reliability problem, the fault-free operation remains the common case and must be optimized.

These insights drive the design of a novel reliability solution that employs near zero overhead “always-on” monitors to detect hardware faults by watching for anomalous software behavior (called symptoms). After a detection, a potentially expensive diagnosis algorithm is invoked to diagnose the source of the error and ensure full recovery. While the diagnosis may incur high overhead, it is only invoked in the rare case of a detection. We believe that the very low cost detection coupled with higher cost diagnosis is the right tradeoff for achieving very low cost reliability solutions.

With these strategies, this dissertation presents a comprehensive reliability solution, called *SWAT* (SoftWare Anomaly Treatment), that detects, diagnoses, and recovers from in-field faults at very low cost. For hardware error detection, SWAT relies on low cost, “always-on” monitors of software symptoms. After a detection, SWAT uses a novel technique called trace based fault diagnosis to identify whether the symptom detection is a result of a hardware or software error and to diagnose the faulty microarchitectural component in case

of a permanent hardware fault. For recovery, SWAT aims to leverage current techniques that use hardware checkpointing for restoring the fault-free execution state coupled with output event buffering for preventing hardware faults from propagating and becoming visible outside of the system.

We evaluated the SWAT system with statistical fault injection experiments. Our results show that simple monitors of software symptoms can achieve high hardware fault detection coverage for permanent and transient faults. After a detection, our trace-based microarchitecture-level diagnosis correctly identifies most of the detected permanent hardware faults, facilitating fine-grained repair. For recovery, although we did not propose a new recovery scheme, we found that high system recoverability can only be attained by employing both hardware checkpointing and output buffering mechanisms, and we identify the overheads for each.

The final contribution of this dissertation is to investigate the accuracy of microarchitecture-level fault modeling. To achieve this goal, we present a novel fault simulation framework called SWAT-Sim that can model gate-level faults accurately while achieving speed comparable to microarchitectural simulations. Using SWAT-Sim, we found that existing microarchitecture-level fault models exhibit some inaccuracies when representing gate-level faults. The SWAT-Sim framework, therefore, serves as an important research vehicle for both SWAT and other ongoing or future research in hardware reliability.

In summary, this dissertation shows, for the first time, that a comprehensive low-cost hardware reliability solution can be realized by treating the software-level symptoms caused by both permanent and transient hardware faults. The presented work lays the foundation for the SWAT approach and paves the way for future work on low-cost software anomaly based resilient systems.

Acknowledgments

Ever wonder why gratitude is neither countable nor measurable? I believe that had gratitude been quantifiable, I would be able to repay all the people that had helped me throughout my doctoral study, and I would not feel so indebted to each and every one of them. This dissertation is dedicated to the many people that have supported me both directly and indirectly throughout the years.

First and foremost, I would like to express my deepest gratefulness to Prof. Sarita Adve. On that cold November day, she offered a research opportunity to a young man that had near-zero background in computer architecture research. Throughout the years, I have grown, unknowingly, into a capable independent researcher because of her generosity, because of her attention to detail, because of her invaluable insight, because of her commitment to excellence, and because she understands what it is like to be a graduate student. For this, I wish I could have a way to pay her back.

I am very thankful to my dissertation committee. Prof. Marc Snir's intriguing comments had sharpened this work immensely. Prof. Craig Zilles had offered much insight that guided this work in its current form and selflessly shared his view on the road to academia. Dr. Pradip Bose, I cannot thank you enough for your guidance on this work since its infancy. This project would have never started without the tremendous effort by Prof. Yuanyuan Zhou and Prof. Vikram Adve. Prof. Yuanyuan Zhou had discussed patiently with us on issues related to operating systems and software reliability and offered invaluable advice throughout the development of SWAT. It is from Prof. Vikram Adve that I have learned why compiler technologies are keys to the success of hardware-software co-design systems. I cherish all the discussion, technical or not, with him.

Many people have offered help throughout my graduate school years and I would like to extend my thanks to them. Dr. Ruchira Sasanka, Dr. Yen-Kuang Chen, and Dr. Eric Debes had been great mentors that helped me grow quickly when I was a junior graduate student. My internship at Intel remains the best

industry experience I have ever had because of the guidance under Dr. Chris Hughes and Dr. Yen-Kuang Chen.

Without the many friends in SC-4111, my life as a graduate student would have been much more dull. I give my many thanks to Pradeep Ramachandran, who co-led various parts of the SWAT project. Over the years, we have become the dream team in both SWAT and badminton. I am very glad to have a friend like you. I would also like to thank Siva Hari, who is never short of great research ideas. I enjoyed the various interesting discussions we have had. I am also glad to know Byn Choi, with whom I always struck good conversation on different topics. Further, I am also thankful to have known all other members of the RSIM group: Rakesh Komuravelli and Hyojin Sung.

Lastly, I am forever indebted to my parents, who always supported me in every possible way throughout my graduate study. Saying thank you is just not enough for my wife, Ting, who has done nothing but cheering me on ever since the first second we met. Amber, my dearest gem, I cannot tell you how much joy you have brought me that thrusts this work to completion.

Table of Contents

List of Tables	x
List of Figures	xi
Chapter 1 Introduction	1
1.1 Motivation and Objectives	1
1.1.1 The SWAT Error Resilient System	2
1.2 Contributions	4
1.2.1 Detection Using Software-Level Symptoms	5
1.2.2 Diagnosis of Permanent Faults by Analyzing Instruction Traces	5
1.2.3 Recovery of Faults through Checkpoint/Replay and Output Event Buffering	6
1.2.4 Accurate System-Level Simulation of Permanent Hardware Faults	6
1.3 Organization	7
Chapter 2 Related Work	8
2.1 Fault Detection	8
2.2 Fault Diagnosis	10
2.3 Error Recovery	10
2.3.1 Hardware-Based Error Recovery	10
2.3.2 Software-Based Backward Error Recovery	11
2.3.3 Input/Output Event Handling	12
Chapter 3 Overview of the SWAT System	13
3.1 Functional Operation of the SWAT System	13
3.2 Components of the SWAT System	15
3.2.1 Detection	15
3.2.2 Diagnosis	16
3.2.3 Repair	17
3.2.4 Recovery	17
3.3 Advantages of the SWAT System	18
3.4 Summary	20
Chapter 4 SWAT Detection	21
4.1 Hardware-Only Software Anomaly Monitors	21
4.1.1 Fatal Traps	22
4.1.2 Hangs	22

4.1.3	High-OS Activity	24
4.1.4	Kernel Panic	24
4.2	Software-Assisted Software Anomaly Monitors	25
4.2.1	Range-Based Likely Program Invariants	25
4.3	Handling False Positives	27
4.4	Methodology – Base Environment	28
4.4.1	Base Simulation Environment	28
4.4.2	Fault Models	31
4.5	Methodology – Detection	33
4.5.1	Symptoms Studied	33
4.5.2	Fault Simulation Experimental Setup	35
4.5.3	Metrics	36
4.6	Results – Hardware-Only Detectors	38
4.6.1	Detection Coverage	38
4.6.2	Software Components Corrupted	47
4.6.3	Detection Latency	49
4.6.4	Transient Faults	50
4.7	Results – Software-Assisted Detectors	52
4.7.1	False Positives	53
4.7.2	Coverage	55
4.7.3	Latency	56
4.7.4	Overhead	57
4.8	Summary and Discussion	58
Chapter 5	SWAT Diagnosis	60
5.1	Diagnosis Overview	62
5.2	Diagnosing Software Bugs, Transient Hardware Faults, and Permanent Hardware Faults	63
5.3	Diagnosing at the Microarchitecture Level	66
5.3.1	Test Trace Generation	67
5.3.2	Test Trace Analysis	69
5.3.3	Implementation	74
5.3.4	Alternative Strategy for TBFD	77
5.4	Methodology	78
5.4.1	Faults Diagnosed	78
5.4.2	Implementation Assumptions	78
5.5	Results	80
5.5.1	Summary of Diagnosis Coverage	80
5.5.2	Uniquely Diagnosed Faulty Structures	81
5.5.3	Non-Uniquely Identified Faulty Structures	82
5.5.4	Faults Diagnosed in Higher Granularity	83
5.5.5	Undiagnosed and Incorrectly Diagnosed Faults	83
5.5.6	Diagnosis Latency	84
5.6	Summary and Discussion	86

Chapter 6	SWAT Recovery	89
6.1	Constraints and Requirements of SWAT Recovery Module	90
6.2	Mechanism for Execution Replay	91
6.3	Checkpoint and Replay Mechanisms	91
6.3.1	Software Checkpointing	92
6.3.2	Hardware Checkpointing	93
6.4	Input/Output State Buffering and Recovery	97
6.5	Exploration of Checkpoint Recovery and I/O Buffering Methods	98
6.6	Methodology	100
6.6.1	System Recoverability	101
6.6.2	Performance Overhead of Hardware Checkpointing	103
6.6.3	Storage Overhead of Output Buffering	104
6.7	Results	106
6.7.1	System Recoverability	106
6.7.2	Ensuring Full System Recovery in SWAT	108
6.7.3	Performance Overhead of Hardware Checkpointing	110
6.7.4	Storage Overhead of Output Buffering	112
6.7.5	Overall Recovery Scheme	117
6.8	Summary and Discussion	119
Chapter 7	SWAT-Sim: Fast and Accurate Simulation of Permanent Hardware Faults	121
7.1	Background	121
7.2	The SWAT-Sim Infrastructure	124
7.2.1	Interfacing the Simulators	124
7.2.2	Different Microarchitecture-Level Structures	126
7.3	Methodology	127
7.3.1	SWAT-Sim Environment	127
7.3.2	Fault Models	128
7.3.3	Parameters of the Fault Injection	128
7.3.4	Studying System-Level Effects	129
7.3.5	Limitations of the Evaluation	129
7.4	Results	130
7.4.1	Performance Overhead	130
7.4.2	Accuracy of Microarchitecture-Level Fault Models	132
7.4.3	Differences Between Fault Models	136
7.4.4	Probabilistic Microarchitecture-Level Fault Models	141
7.5	Summary and Discussion	144
Chapter 8	Conclusions and Future Work	147
8.1	Conclusions	147
8.2	Limitations and Future Work	149
8.2.1	Fundamentals of Symptom-Based Detection and Application-Aware Metrics for Evaluation	149
8.2.2	Implementation of SWAT	150
8.2.3	Hardware Fault Models	151
8.2.4	Multithreaded Workloads on Multicore Systems	152

8.2.5 Recovery Mechanisms	153
References	156
Author's Biography	162

List of Tables

4.1	Parameters of the simulated processor.	29
4.2	Description of server workloads.	31
4.3	Microarchitectural structures in which faults are injected. In each run, either a stuck-at fault is injected in a random bit or a bridging fault is injected in a pair of adjacent bits in the given structure.	31
6.1	Comparison of hardware and software checkpointing schemes.	92
6.2	Parameters of the simulated multicore system.	103
7.1	Slowdowns of SWAT-Sim when compared to pure μ arch-level simulation.	131
7.2	Percentage of bits incorrect at the output latch.	138

List of Figures

3.1	Operation of the SWAT system.	14
3.2	Functional overview of the SWAT system.	15
4.1	Hardware structure used for hang detection.	23
4.2	Simulation environment. (a) A single-system environment that runs SPEC applications on a commercial OS. (b) A two-system environment that runs server applications on one system and client applications on another system.	30
4.3	Outcomes of an injected fault. If the injected fault is not detected within 10M instructions, the fault is removed (no new fault activation, but software state may already be corrupted at this point) and the application is functionally simulated to completion to identify its effect on the application's outputs or whether it causes a detected unrecoverable error (DUE). . . .	36
4.4	Coverage of SWAT hardware-only detectors for (a) SPEC workloads for both stuck-at and bridging permanent faults and (b) server workloads for stuck-at permanent faults.	39
4.5	Distribution of detections by fatal traps for SPEC workloads. The <i>Other</i> category constitutes Data Access Exception, Protection Violation and Division by Zero traps, which make up <8% of detections by fatal traps. The total height of a bar is the percentage of the injected faults in the corresponding structure that caused fatal hardware traps.	43
4.6	Distribution of detections by fatal traps for server workloads. The <i>Other</i> category constitutes Data Access Exception, Protection Violation and Division by Zero traps, which make up <2% of detections by fatal traps. The total height of a bar is the percentage of the total injected faults in the corresponding structure that caused fatal hardware traps.	44
4.7	Application and system state integrity for the detected faults in (a) SPEC workloads and (b) server workloads. The height of each bar gives the percentage of injected faults detected in that structure. We see that most faults corrupt the system state.	48
4.8	Total number of instructions retired from architectural state corruption to detection for SPEC workloads.	50
4.9	Total number of instructions retired from architectural state corruption to detection for server applications.	51
4.10	Coverage of SWAT hardware-only detectors for SPEC workloads on transient faults.	52
4.11	Coverage of SWAT hardware-only detectors for server workloads on transient faults.	53
4.12	Variation of False positives rate with different number of training inputs. The rate is <5% with 12 training sets, motivating the use of 12 inputs for the rest of our experiments.	54
4.13	Permanent fault coverage of hSWAT and iSWAT.	55

4.14	Detection latencies for hSWAT and iSWAT. The percentages are computed using number of recoverable detections in iSWAT as baseline. The invariants increase the number of faults detected within 1,000 instructions by 2%.	56
4.15	Overhead of invariants on an UltraSPARC-IIIi (Sparc) machine and an AMD Athlon machine (x86).	57
5.1	Diagnosis of a detected symptom. Through repeated replays, a software bug, a transient, or a permanent hardware fault is diagnosed.	64
5.2	Diagnosis of a permanent hardware fault. By comparing the fault-free and faulty executions and analyzing the resulting test trace, the faulty microarchitectural unit is diagnosed.	67
5.3	An example scenario depicting how a physical register that is mapped to more than one logical register is identified by TBFD.	73
5.4	An example Instruction Trace Buffer (ITB). For each instruction retired by the faulty core during trace-based diagnosis, the ITB records information pertaining to 1) decoded instruction information, 2) some microarchitectural resources used by the instruction, and 3) the data values used by the instruction.	74
5.5	Effectiveness of microarchitecture-level fault diagnosis. The figure shows the ability of the diagnosis algorithm to accurately diagnose detected faults. Overall, 98% of the detected faults are accurately diagnosed as either (1) the correct non-array structure or the correct entry within an array structure (the Unique stack); or (2) within one of two non-array structures or entries of array structures (Among 2); or (3) the correct array structure type but not the correct entry within the structure (Correct Type).	81
5.6	Diagnosis latency in number of instructions executed by the faulty core between the start of diagnosis and the point when the fault is diagnosed. The figure shows that over 90% of the faults can be diagnosed within 1 million instructions.	85
6.1	Recoverability of server workloads	107
6.2	Slowdowns in fault-free execution of (a) LU, (b) FFT, (c) Radix, and (d) Ocean due to hardware checkpointing.	111
6.3	Maximum number and size of CPU-to-Device requests. (a) Maximum number of stores issued to the device by the CPU for varying buffering intervals. (b) Maximum buffer size for storing the CPU-to-Device write requests (address and data) for different buffering intervals.(c) Maximum number of loads issued to the device by the CPU for varying buffering intervals. (d) Maximum buffer size for storing the CPU-to-Device read requests (address and data) for different buffering intervals.	114
6.4	Interaction between CPU and specific devices in Apache and SSH daemon. (a) and (b) shows the maximum buffer size for storing device-specific writes and reads, respectively, in Apache. (c) and (d) shows the maximum buffer size for storing device-specific writes and reads, respectively, in SSH daemon.	115
7.1	Comparison of how a faulty μ arch-level unit X is simulated by (a) a pure μ arch-level simulator and (b) by SWAT-Sim.	125
7.2	Efficacy of the SWAT fault detection scheme under different fault models for the ALU, AGEN, and Decoder. Depending on the fault model and the structure, the μ arch-level fault may or may not capture the system-level effects of gate-level faults accurately, as indicated by the differences in coverage.	133

7.3	Latency of fault detection in terms of number of instructions executed from architectural state corruption to detection. The differences in the models impact recovery, which is primarily governed by these latencies.	135
7.4	Mean fault activation rate for the different fault models as a percentage of the number of instructions.	136
7.5	Probability of corrupting each bit of the ALU output latch, under μ arch-level s@0, gate level s@0, and gate level delay models.	139
7.6	Probability of corrupting each bit of the AGEN output latch, under μ arch-level s@0, gate level s@0, and gate level delay models.	140
7.7	The accuracy of the derived P- and PD-models for modeling gate level faults in ALU, evaluated using the coverage of the SWAT detectors.	142
7.8	The accuracy of the derived P- and PD-models for modeling gate level faults in AGEN, evaluated using the coverage of the SWAT detectors.	143

Chapter 1

Introduction

1.1 Motivation and Objectives

For decades, the number of transistors integrated on-chip has grown faithfully according to Moore's Law. This exponential growth, thanks to the advance of CMOS process technology that allows the ever-decreasing device size, provides opportunities for chip makers to introduce microprocessors with higher levels of system integration that provide increased computing capability. On the other hand, while modern computer systems continue to reap the benefits of continued device scaling, there is a growing concern for the reliability of these systems. As the level of system integration continues to increase, a chip containing a growing number of shrinking devices (e.g., Intel Itanium Tukwila processors contain 2 billion transistors [28]), statistically, is expected to experience more device failures. This growing reliability threat is recognized by the industry as one of the grand challenges for designing future computer systems in the International Technology Roadmap for Semiconductors (ITRS) [1]. In particular, future devices are expected to fail in the field for a variety of reasons such as wear-out or aging, soft errors caused by radiation, process variation, infant mortality, design defects, and so on [8]. Hence, it is highly desirable to have a general reliability solution that detects, diagnoses, recovers from, and repairs around components that fail in the field for a multitude of reasons.

Since the problem of building fault-tolerant systems is not new, one may argue that traditional solutions involving redundancy in space, time, and/or information [48, 68] can be leveraged to neutralize this pending reliability threat. The difference between the traditional reliability problem and the current growing threat caused by device scaling, however, lies in the affected segments of computing markets. Traditionally, hardware reliability mainly concerned high-end niche systems such as transaction processing systems for banks and mission-critical systems for space applications. As the main priority for designing these systems is to meet reliability goals, the budget spent on ensuring reliability is less constrained than mainstream systems

and the use of solutions that involve heavy amount of redundancy, such as triple modular redundancy, is acceptable.

On the other hand, the scaling-induced reliability problem is fundamentally different. Most, if not all, computer systems have taken advantage of the ever-growing system integration made possible by device scaling and will continue to do so. The reliability problem caused by scaling, therefore, will pervade most of the computing market and hardware reliability will be a concern even for the mainstream computer systems. In these market segments, because the budget that can be spent on reliability is much more limited than that of the high-end systems, an effective solution must incur low overheads in area, power, and performance in order to be broadly deployable; this precludes the use of traditional solutions that rely on expensive redundancy. To put it into perspective, an industrial panel in a recent workshop converged on a 10% area overhead target to handle all sources of chip errors as a guideline for academic researchers [72].

With this pending reliability threat, the research challenge is to derive a low cost yet effective reliability solution that can cater to the mass computing market. Driven by this reliability trend, recent research has focused on deriving low-cost reliability solutions (e.g., [4, 13, 45, 61, 54, 73, 81]). This dissertation also investigates such a low-cost reliability solution. However, while these previously proposed schemes have either focused on low-cost detection mechanisms or detection and recovery mechanisms for transient faults, we take a holistic system design approach to derive a complete reliability solution including detection, diagnosis, and recovery that handles both permanent and transient faults. (Chapter 2 gives a more detailed comparison of our approach with prior work.)

1.1.1 The SWAT Error Resilient System

The main contribution of this dissertation is the proposal of a novel low-cost solution for hardware reliability. There are two key observations that motivate our design approach.

- First, a hardware fault is only considered harmful if it affects software execution. Hence, an effective reliability solution only needs to handle hardware errors that propagate through high levels of the system and become observable to the software.
- Second, even though the reliability threat is growing, fault-free operation still remains the common case. Therefore, reliability solutions must be optimized for fault-free operation.

Based on these observations, we follow a design philosophy that minimizes the total cost of the system by minimizing the overhead of fault-free operation as much as possible, at the expense of higher cost paid for the uncommon case (not unlike Amdahl’s Law). In a fault-tolerant system, as the error detection mechanisms need to be *always on*, our design focuses on minimizing the overhead of the detection component. This is achieved by allowing hardware errors to propagate to the software level and detecting them through monitoring the abnormal software behavior (called *symptoms*) using zero to very low overhead hardware and/or software monitors. After a detection occurs, the diagnosis algorithm is invoked to diagnose the source of the error. While the potentially long latency of high-level symptom-based detection can make the diagnosis more complex and expensive, we believe this is the right tradeoff because diagnosis is only invoked after a *rare event* of a detection.

These strategies motivate the design of a comprehensive reliability solution, called *SWAT* (SoftWare Anomaly Treatment), that detects, diagnoses, recovers from, and repairs/reconfigures (in the case of a permanent hardware fault) around failed components in the field. SWAT relies on low-cost symptom monitors, implemented in either hardware or software, for detecting hardware faults that manifest into the software and cause anomalous software behaviors. After a detection, the diagnosis procedure, controlled by a thin layer of firmware, exploits repeated rollbacks/replays in the multicore environment for diagnosing the source of the error. To recover from an error, SWAT employs a checkpoint/replay mechanism to roll back the faulty execution to the previous fault-free checkpoint and an output event buffering mechanism to prevent the effect of the fault from propagating to the outside world. In case of a diagnosed permanent fault, the failed component needs to be repaired for full recovery. SWAT relies on existing built-in redundancy in modern superscalar processor to reconfigure around (e.g., disabling one of the integer ALUs) the failed unit.

While SWAT primarily relies on symptom monitors for error detection, the SWAT approach naturally extends to incorporate backup detection techniques (e.g., hardware checkers, selective redundancy, online test) for the cases where the high-level symptom-based detection coverage is determined to be insufficient; e.g., for some mission-critical applications or in case of some faults in some structures that may not easily reveal detectable symptoms at the required cost. Compared to any one such technique used in isolation, the SWAT approach has the following advantages (discussed in detail in Chapter 3).

- Total system cost is minimized by focusing on optimizing for the common-case error detection mech-

anisms.

- The high-level symptom-based detection mechanism is general and does not tie to a specific-failure mode (e.g., soft errors), making it extensible for other failure mechanisms (even ones that are not yet known).
- Faults that are masked in various levels of the hardware systems are naturally ignored by the SWAT detection mechanisms, avoiding excessive overheads. Further, even hardware faults that are masked in the application software are correctly ignored, since they do not appear as software anomalies.
- As the SWAT system is controlled and coordinated by a thin firmware layer, it can be customized to match different application-specific and system-specific reliability needs.
- By taking a holistic system design approach, novel solutions can be derived. For example, diagnosis can rely on the rollback/replay recovery mechanism to precisely diagnose the symptom-causing errors.
- The symptom based detection mechanisms are essentially detecting software bugs. This presents an opportunity to explore the use of software bug detection techniques to ensure hardware reliability, amortizing the overhead for different system functions. In the long term, the SWAT system can evolve to provide a unified framework for both hardware and software reliability.

While SWAT has many advantages when compared to prior work, it also has certain limitations. We discuss the limitations and future work of SWAT at the end of this thesis. However, recent work along with my colleagues (not reported here) has already addressed several important issues in SWAT (e.g., [26]); we briefly discuss this at the end of the thesis. Overall, this thesis provides the foundation for the SWAT approach and paves the way for much of the ongoing and future work. In the long term, we believe that the SWAT approach is key to provide a unified framework for both hardware and software reliability.

1.2 Contributions

The following summarizes the contributions of this dissertation in greater detail.

1.2.1 Detection Using Software-Level Symptoms

We conducted both permanent and transient fault injection experiments and employed a number of software anomaly symptom monitors to detect the injected faults [36]. For permanent faults, our results show that (1) simple symptom detectors that incur zero to little hardware overhead are able to detect 98% and 99% of the unmasked faults in 7 studied microarchitectural structures for SPEC workloads and server workloads, respectively, (2) a large fraction of detections corrupt OS state for both SPEC and server workloads, motivating the needs for OS recovery, and (3) while all of the detections have latencies that are short enough so that the pristine execution state can be efficiently restored using hardware checkpointing schemes, full system recovery still depends on whether I/O activities can be properly buffered to prevent fault propagation to the outside world. For transient faults, our results show that (1) 96% and 90% of the faults are masked for SPEC and server workloads, respectively, (2) 59% of the unmasked faults are detected by our symptom detectors for both SPEC and server workloads, which is consistent with previously proposed symptom-based transient fault detection schemes [61, 81].

We also explored using likely program invariants, a well-known bug detection method, to detect permanent hardware faults [70]. (This work was led by Swarup K. Sahoo.) When used with the simple symptoms described above, likely invariants are able to reduce the silent data corruption (SDC) events by 73% when compared to a system that uses only the simple symptom monitors.

These results clearly show that monitoring software-level misbehavior is effective in detecting permanent and transient hardware faults.

1.2.2 Diagnosis of Permanent Faults by Analyzing Instruction Traces

After an error detection, SWAT must diagnose the source of the fault to ensure full recovery. We propose a diagnosis framework that exploits rollback/replay in a multicore environment to (1) distinguish among software bugs, transient hardware faults, and permanent hardware faults and (2) diagnose the microarchitectural component that contains the permanent fault by comparing and analyzing the faulty and fault-free instruction traces with a technique called trace-based fault diagnosis (TBFD) [35]. We found that TBFD is able to diagnose 98% of the faults detected by SWAT and 90% can be exactly diagnosed to an array entry or a non-array unit. These results show that hardware permanent faults are highly diagnosable through instruction trace

analysis.

1.2.3 Recovery of Faults through Checkpoint/Replay and Output Event Buffering

For error recovery, we attempt to recover all faults injected into the I/O intensive server workloads that are detected within 10 million instructions. For this purpose, similar to other recent work, we leverage existing techniques that perform hardware checkpointing [74, 59]. Our contribution here is in quantitative results that show that both hardware checkpointing and output buffering mechanisms are required for full recoverability and in determining the overheads for each, for the detection latencies of SWAT. We find that with long checkpoint intervals, the checkpointing mechanism (Revive on multicore) degrades performance only slightly but the output buffering mechanism requires somewhat larger storage. Conversely, with short checkpoint intervals, the storage needed by the buffering mechanism is much smaller, but the overhead of the checkpointing mechanism increases. Thus, although the overheads with the current techniques are manageable, they are not as low as for the rest of the SWAT system. These results motivate more efficient recovery schemes that find a lower-overhead sweet spot for both checkpointing and output buffering overheads, possibly enabled by a further reduction in detection latencies for SWAT. We leave this exploration to future work.

1.2.4 Accurate System-Level Simulation of Permanent Hardware Faults

Most of this thesis uses microarchitecture-level fault models to represent hardware faults. These fault models, however, are potentially inaccurate as hardware faults occur at a lower level. Because there were no other methods that model hardware faults at a lower level and capture their impact on software, microarchitecture-level fault models were used.

To address this issue, our final contribution is SWAT-Sim, a fast and accurate hierarchical fault simulator, for observing how gate-level permanent faults in the combinational logic propagate to the system level [34]. SWAT-Sim achieves speed by simulating mostly at the microarchitecture level and invoking the gate-level simulation only when the faulty component is used. We found that SWAT-Sim is 100,000x faster than gate-level simulations but only 3x slower than microarchitectural simulations while maintaining gate-level fault modeling fidelity. We use the results from SWAT-Sim to understand the accuracy of the microarchitecture-level fault models for the ALU, AGEN, and Decoder, and found that (1) the accuracy of the

microarchitecture-level stuck-at models, while is dependent on different structures, is inadequate for some cases, (2) the activation rates and the bit corruption patterns vary significantly between the microarchitecture-level and the gate-level fault models, attributing to the different fault behaviors, and (3) our attempt to derive probabilistic microarchitecture-level fault models using data from SWAT-Sim for gate-level faults remains unsuccessful. While these results do not change the qualitative findings for the detection, diagnosis, and recovery modules, they imply that hardware fault injection based studies of current and future hardware reliability solutions, including SWAT, should consider using techniques like SWAT-Sim to attain more accurate evaluations.

1.3 Organization

The rest of this dissertation is organized as follows. Chapter 2 discusses the work related to the SWAT detection, diagnosis, and recovery schemes. Chapter 3 presents an overview of the SWAT system. Chapter 4 discusses the very low cost symptom-based detection scheme in SWAT and shows the effectiveness of the SWAT detection approach through fault injection experiments. Chapter 5 presents the SWAT diagnosis algorithm that can effectively isolate the different sources of errors and identify the diagnosed permanent hardware faults at the microarchitecture level. Chapter 6 investigates SWAT recovery. Chapter 8 concludes this dissertation and discusses future research directions of the SWAT error resilient system.

Chapter 2

Related Work

While there have been several recent (relatively) low-cost hardware reliability proposals that eschew excessive redundancy, they have dealt with detection, diagnosis, and recovery as independent problems and/or they typically propose fault-specific solutions that may not work for other fault types. To the best of our knowledge, SWAT is the first solution that overcomes such shortcomings, yielding a generic full-system solution at low cost.

The following sections briefly describe these related studies.

2.1 Fault Detection

As mentioned in Chapter 1, our focus is on low-cost reliability for a broader market, where some parts of the market may even be willing to trade off some coverage for cost. There has been substantial microarchitecture level work in this context, where redundancy is exploited at a finer microarchitectural granularity. While much of that work handles transient hardware faults [4, 22, 23, 66, 67, 68, 81], recently, there has been a growing body of work on handling permanent hardware faults. We discuss some of these schemes in the following.

Checker and online-testing based detection.

Austin proposed DIVA, an efficient checker processor that is tightly coupled with the main processor's pipeline to check every committed instruction for errors [4]. While DIVA can be used to provide detection of hard (and transient) errors, it does not provide mechanisms for diagnosis or repair. Bower et al. introduced a hard error diagnosis scheme in the DIVA checker architecture that identifies hard faults through tallying the different structures utilized by the faulty instructions [10]. (We describe the diagnosis algorithm in greater detail in Section 2.2.)

Shyam et al. recently proposed online testing of certain structures in the microprocessor for detecting hard faults, and recovering the system by both disabling the faulty units and rolling back to a hardware checkpoint [73]. Since these tests are run only when the structures are idle, the performance overhead is rather small. Constantinides et al. enhanced this scheme further in [13] by adding hardware support so that the software can control the online testing process, adding flexibility for choosing test vectors. However, the performance penalty incurred by software-controlled online testing is high for reasonable hardware checkpointing intervals. Furthermore, the continuous testing of hardware can accelerate the wear-out process.

As part of the Argus reliability scheme, Meixner et al. have proposed to use computation checkers (information redundancy) to protect the ALUs, multipliers, and dividers from transient and permanent hardware faults in simple cores [45].

All of the above schemes incur significant overhead in area, performance, power, and/or wear-out that is paid almost all the time. Further, these are customized solutions for hardware reliability. In contrast to the above, we seek a reliability solution that pays minimal cost in the common case where there are no errors. In other words, we seek an “always-on” error detection mechanism that has minimal cost in area, performance, and power.

Software-centric detection.

There is a large body of literature on detecting hardware faults through monitoring software behavior [22, 51, 54, 61, 65, 67, 79, 81]. The majority of this work focuses on control flow signatures, crashes, and hangs. Recent work has also examined value based invariants extracted in hardware [61], invariants in software that are extracted ahead-of-time [54], and locality of instruction-level invariants [17] for detecting errors. These schemes are similar to our more sophisticated software-assisted detection scheme (to be discussed in Chapter 4).

Other low-cost hardware-based fault detection schemes have also been proposed. Meixner et al. have proposed the use of data and control flow checkers for transient and permanent faults in simple single-issue, in-order pipelines, with no interrupts [45]. Our proposed symptom-based detectors work at a much higher level – they are largely oblivious to the microarchitecture and require very little hardware overhead.

2.2 Fault Diagnosis

Online diagnosis of hardware faults is a relatively new research area when compared to the detection and recovery parts of the fault-tolerant systems. We discuss some of the recent proposals here.

The online testing schemes [73, 13] discussed earlier assume both the roles of error detection and fault diagnosis. If a particular hardware module fails a particular test, that module is instantly diagnosed as faulty and can be taken offline. However, these online tests are generated based on pre-specified fault models. New unknown failure modes may cause a fault to go undetected and undiagnosed. Our diagnosis scheme aims to replicate the same execution environment online so that these faults can be repeatedly activated and correctly diagnosed.

The most related prior work to our diagnosis scheme (discussed in Chapter 5) is by Bower et al. [11], proposed in the context of the DIVA architecture [4]. Their scheme associates a counter for each reconfigurable (repairable) microarchitectural resource. As instructions flow through the pipeline, it keeps track of the microarchitectural resources used (e.g., which ALU, etc.) in a bit vector which is carried along through the pipeline. When a mismatch between the main processor and the DIVA checker is detected, the counter corresponding to each resource utilized by the mismatching instruction is incremented. Once a resource counter reaches a certain threshold value, it is diagnosed with a permanent hardware fault. Our scheme differs from that of Bower et. al. in the following ways. First, we incur diagnosis related overhead only in the infrequent case when a fault is detected. Their scheme, however, contains always-on monitors (for diagnosis) that present overheads in power and performance even in the common fault-free operation. Second, although their method works well for faults on the data path, it is not well-suited to handle faults in structures that establish or rely on logical to physical register name translations. Our scheme diagnoses the faulty microarchitectural structure even in these scenarios.

2.3 Error Recovery

2.3.1 Hardware-Based Error Recovery

As suggested by its name, the hardware-based error recovery methods are ones that have specialized hardware support for recovering detected hardware errors. Hardware-based error recovery schemes can be

broadly classified into forward error recovery and backward error recovery.

Forward error recovery (FER).

These schemes use redundant hardware to detect and correct faults so that forward progress can be guaranteed. Traditionally, high-end systems employ triple modular redundancy [6] to mask the detected fault through voting. More recently, Austin proposed DIVA that uses a checker processor to mask faults in the main processor [4]. Because FER schemes usually involve significant amount of redundancy, they are considered too expensive for the mainstream market and thus not suitable for use in SWAT.

Backward error recovery (BER).

These schemes are more commonly known as rollback-and-replay recovery methods. They generally involve some form of checkpointing (taking a snapshot of the state) or logging (generating an undo log to recover the state) to establish checkpoints, to which the system can be rolled back after an error detection. Traditionally, IBM mainframes [75] contain register checkpoint hardware and store-through caches to recover from processor and memory errors. To reduce the cost of checkpoint creation, there have been various proposals that involve modest enhancements to the processor such as attaching a snooping device for logging [43] and making enhancements to the cache so that the dirty data occupancy in the cache triggers multiprocessor checkpoint establishments [2]. While these schemes offer solutions for early hardware-based checkpoint/rollback recovery, there is little controllability of the checkpoint interval.

Recently, SafetyNet [74] and ReVive [59] were proposed to provide a sophisticated method for taking periodic consistent multiprocessor checkpoints. Because these schemes are closely tied to the design of the SWAT recovery module, we leave this detailed discussion to Chapter 6.

2.3.2 Software-Based Backward Error Recovery

Software-based backward error recovery schemes have also been proposed to improve fault tolerance. They work by periodically establishing checkpoints of the process state so that a failed process can be restarted from the checkpointed state, instead of the beginning of the execution.

In particular, HP NonStop servers have every process periodically checkpoint its state on another processor [6]; the shadow process can take over once the main process fails. Elnozahy and Zwaenepoel [19] introduce Manetho that coordinates different processes in the distributed system to take checkpoints so that

the domino effect is minimized. Plank et al. [58] introduce diskless checkpointing and explore different ways to store the checkpoint data in the main memory to reduce the performance overhead. Flashback [76] relies on a shadow process based checkpoint creation mechanism in diskless checkpointing to provide efficient support for software debugging. BLCR [5], an ongoing project, is a Linux module developed for checkpointing the Linux applications. Because the SWAT recovery module can potentially leverage software-based error recovery methods, we discuss some of these schemes in detail in Chapter 6.

2.3.3 Input/Output Event Handling

Besides checkpointing, input/output commit problems need to be properly handled to prevent inconsistencies in the system that may thwart full recovery. BLCR [5] and Flashback [76] offer partial solutions for the I/O recovery problem (e.g., recovering file I/O) by checkpointing the I/O information (e.g., file handle) in the kernel-specific data structures. Nakano et al. propose ReViveI/O [50] to buffer disk and network events using pseudo device drivers (PDDs) for fault containment. Compared to BLCR and Flashback, ReViveI/O provides a more general solution that is less dependent on the OS kernel. As the SWAT recovery module also needs a general mechanism to properly handle I/O events, we discuss ReViveI/O in detail in Chapter 6.

Chapter 3

Overview of the SWAT System

The primary goal of the SWAT system is to provide a very low-cost hardware reliability solution for most of the computing market. To achieve this, the design of the SWAT system is based on two key observations.

1. An effective solution only needs to handle hardware faults that propagate into higher levels of the system and corrupt the software execution.
2. Even though the future hardware failure rate is projected to increase, fault-free operation remains the common case and hence must be optimized.

Based on these observations, SWAT minimizes common-case cost by using zero to very low overhead hardware and/or software monitors to detect hardware faults that manifest into the software. SWAT then invokes a potentially expensive diagnosis routine after a rare case of a detection.

In the rest of the chapter, we first describe the functional operation of the SWAT system. Then, we take a closer look at the various components of the SWAT system. After that, we discuss the potential advantages of SWAT.

3.1 Functional Operation of the SWAT System

Figure 3.1 shows the high-level view of the typical operation of the SWAT system. For error recovery, SWAT relies on a form of checkpoint/replay mechanism to roll back the system to a pristine state. Hence, in the figure, the checkpoints are created periodically. Further, because the fault-handling operations are non-trivial, SWAT relies on a thin firmware layer (not shown in the figure) to coordinate detection, diagnosis, recovery, and repair.

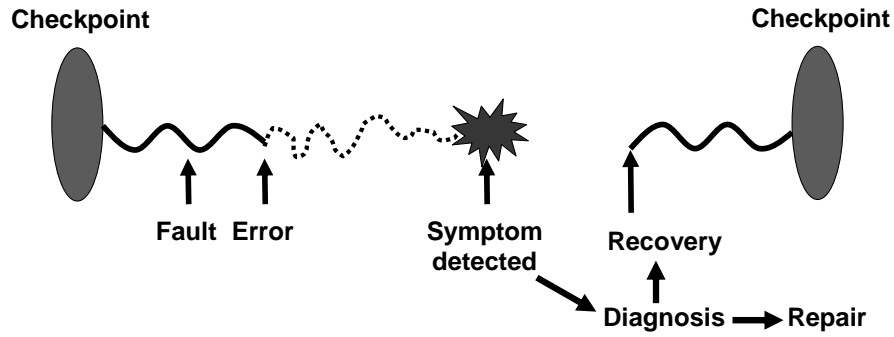


Figure 3.1: Operation of the SWAT system.

The SWAT system operates as follows. From the figure, the software continues to execute after a checkpoint is created (shown as the solid wavy line). Some time later, a hardware fault appears. Because this fault has not been activated by the software, the execution is still correct. As the software execution continues, this hardware fault is exercised and an error is introduced to the software. From this point on, the software execution is potentially incorrect. This error then continues to manifest in the software execution (shown as the dotted line). If the underlying fault is a permanent hardware fault, multiple fault activations can occur and result in multiple corruptions in the software execution. Eventually, the error(s) causes a symptom that is detected by one of the symptom monitors in SWAT. Now, the hardware fault appears as a form of software anomaly and is considered detected.

After a detection, to treat this software anomaly, the SWAT firmware is invoked to coordinate all the fault-handling operations. The first step is to diagnose the cause of the detected symptom. The SWAT diagnosis algorithm currently assumes three different fault models: software bugs, transient faults (either hardware or software), and permanent hardware faults. It also assumes a multicore system. At a high level, SWAT diagnosis watches for symptom re-occurrence on repeated rollbacks/replays on the same or different cores to determine the different sources of errors. If there is no symptom after a simple rollback/replay on the original symptom-causing core, a transient fault is diagnosed and the rollback/replay naturally recovers this type of fault. If the symptom is persistent on the original and a different core in the system, the diagnosis algorithm identifies a deterministic software bug and SWAT propagates the symptom to higher levels of software. However, if the symptom is only persistent on one core but not the others, a permanent hardware fault is diagnosed. Since a hardware fault is persistent, it cannot be recovered through rollback/replay. To

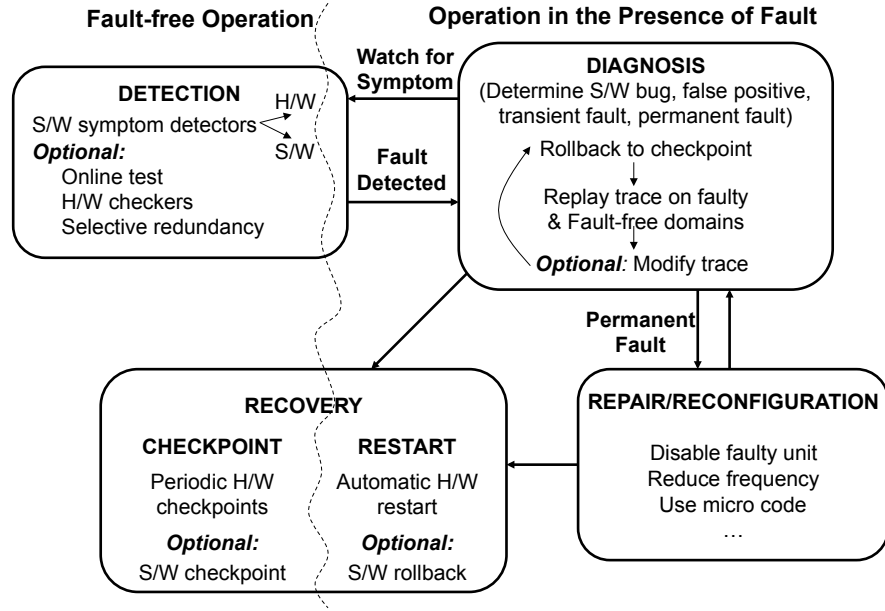


Figure 3.2: Functional overview of the SWAT system.

prevent further corruptions by the underlying permanent hardware fault, SWAT diagnoses the permanent fault at the microarchitecture level to facilitate fine-grained repair or reconfiguration, essentially extending the lifetime of the faulty core. To recover from a detected hardware faults (either transient or permanent), the SWAT firmware invokes the rollback recovery procedure so that the pristine execution state is restored and the correct software execution can be resumed.

3.2 Components of the SWAT System

Figure 3.2 illustrates how the detection, diagnosis, recovery, and repair components work together in SWAT. In the following, we briefly discuss the different modules of SWAT.

3.2.1 Detection

As shown in Figure 3.2, the detection mechanism is always on during both fault-free and fault-handling (during diagnosis) operations. Because the fault-free operation is the common case, the cost paid in detection directly impacts the overall system cost. Focusing on minimizing the cost of the detection module is the primary reason why SWAT is able to achieve low overall system cost.

Based on our previous observations, the sweet spot of handling hardware faults is at the level where the faults have manifested and appeared as some form of software misbehavior. Thus, to minimize the cost of the detection module, SWAT relies on very low overhead symptom monitors to detect the software anomalies caused by the propagation of hardware faults. As some of these monitors have near zero overheads, the cost of the detection module is truly minimized. Further, with monitors that detect anomalous software behavior, they naturally handle only hardware faults that matter and ignore those that do not. For example, if a hardware fault propagates but has its effect masked by a branch misprediction induced pipeline flush, the error is invisible to the software and the symptom monitors. In this work, we first experimented with very low cost hardware-only detectors that do not require assistance from software. Since SWAT treats the hardware errors when they appear as software bugs, many techniques from the software debugging research community can potentially be leveraged. In particular, we also looked into software-assisted detectors that rely on compiler support.

For systems that require higher detection coverage, techniques such as online testing, selective redundancy, hardware checkers, etc. can also be incorporated into the SWAT detection module. The resulting system will be more expensive but is more reliable.

3.2.2 Diagnosis

Post detection, the first step of the fault-handling operation is to diagnose the source of the error. Because a detection is typically rare, the invocation of the diagnosis procedure is also rare. Consequently, the overheads paid in this operation can potentially be expensive but still do not affect the overall system cost significantly.

Because software bugs, transient hardware faults, and permanent hardware faults can all manifest and lead to symptoms, the diagnosis algorithm has to be made necessarily intelligent to precisely diagnose the source of the error. Towards this end, we assume the SWAT firmware is involved in controlling the diagnosis process. We further assume that a single-threaded application is executing on one core in the multicore system. Then, given SWAT has a checkpoint/replay recovery mechanism, the firmware-controlled diagnosis procedure performs repeated replays of the execution in a multicore environment to determine the cause of the symptom.

The procedure goes as follows. After a symptom is detected for the first time, a rollback/replay is

triggered on the symptom-causing core and the firmware watches for any symptom re-occurrence. If no symptom is found, a transient fault is diagnosed and the execution replay duly recovers the fault. If the symptom persists on the same core, the firmware transfers the checkpoint to another core in the system to replay the execution. If the symptom recurs on this new core, SWAT diagnoses this as a deterministic software bug (because the symptom persists on all cores) and propagates its effect to the higher levels of software. However, if the symptom does not occur on this new core, a permanent fault is diagnosed (since it occurs twice on the original core but disappears when the execution is on the new core).

In the case of a diagnosed permanent fault, SWAT can choose to decommission the entire faulty core as a method of repair. But, modern superscalar processors often contain built-in redundancy that can be exploited for fine-grained repair. Hence, to facilitate this level of repair or reconfiguration, SWAT will attempt to diagnose the location of the fault at the microarchitecture level using a technique called trace based fault diagnosis. This technique, in essence, synthesizes a dual modular redundant execution using two cores from the multicore system to identify the source of the permanent fault. Using divergences between the executions as clues, the trace based diagnosis algorithm intelligently tracks down the source of the hardware fault from the collected execution trace. After that, the appropriate repair procedure is invoked to prevent future activations of the permanent fault.

3.2.3 Repair

As mentioned, in the case of permanent fault, repair or reconfiguration is needed to ensure reliable and continuous operation. To this end, SWAT relies on built-in microarchitectural redundancy, frequency and voltage scaling, and/or microcode-level reconfiguration in modern processors to fulfill this task. We believe that this level of control is typically available for current and future multicore systems. Thus, this thesis assumes the necessary support for repair is in place and does not go into the implementation details of the repair mechanisms.

3.2.4 Recovery

From Figure 3.2, error recovery is active in both fault-free (along with SWAT detectors) and fault-handling operations. Based on our observations, in order to keep SWAT low cost, the fault-free overhead incurred by

the recovery mechanism must be kept low as it is the common case operation. Functionally, the recovery mechanism must be capable of restoring the system state and preventing the effect of the fault from propagating outside of the sphere of recoverability (to be discussed in Chapter 6). In this thesis, we assume a hardware-based checkpoint/replay mechanism as the primary SWAT mechanism for execution state restoration and an output event buffering mechanism to prevent faults from propagating and becoming visible to the outside world.

During fault-free operation, the checkpoint/replay mechanism periodically creates checkpoints of the execution state. The buffering mechanism provides the necessary storage to buffer the output events. To be consistent with the SWAT system design approach, these modules must be designed appropriately to minimize the overheads in performance, area, and power in order not to increase the overall system cost significantly.

During the fault-handling operation, the SWAT firmware invokes the rollback recovery procedure of the checkpoint/replay module and discards the potentially faulty output events in the buffering module. Similar to diagnosis, these operations can be allowed to incur higher overhead since they occur infrequently.

In this dissertation, while we do not propose a new recovery scheme, we quantify the need for both the checkpointing and output buffering mechanisms in terms of system recoverability and identify the overhead incurred by these existing schemes. These results are important as researchers can use them to help derive new lower-cost and more efficient recovery mechanisms.

3.3 Advantages of the SWAT System

The SWAT system is designed as a comprehensive reliability solution from the ground up. To realize a very low cost solution, the SWAT approach emphasizes the absolute minimal cost in always-on error detection by using a very low cost software-level symptom-based detection mechanism. Overall, we believe the SWAT system has the following advantages over existing techniques. (We discuss the limitations of SWAT at the end of this thesis.)

- **Optimizing for the common case.** Total system overhead is significantly reduced by emphasizing minimal detection overhead (which is paid all the time), possibly at the cost of higher diagnosis over-

head (which is paid only in the case of a fault).

- **Generality.** High-level symptom-based detection techniques are largely oblivious to specific low-level failure modes or microarchitectural/circuit details. Thus, in contrast to detection methods that are driven by specific device-level fault models (e.g., wear-out detectors), high-level detection techniques are more general and extensible to numerous failure mechanisms and microarchitectures.
- **Ignoring masked faults.** Previous work has shown that a large number of faults are masked by higher levels of the system such as circuit, microarchitecture, architecture, and application levels [15, 32, 37, 49, 82]. High-level detection techniques naturally ignore faults that are masked at any of these levels, avoiding the corresponding overheads.
- **Customizability.** A firmware controlled system with detection mechanisms driven by software behavior provides a natural way for application-specific and system-specific customization of the reliability vs. overhead tradeoff. For example, when a fault is detected in a video application, the system may consider dropping the current frame computation rather than recovering it. Further, the approach is amenable to selective cost-conscious use of different symptom-based and backup detection techniques.
- **Novel solutions result from holistic system design.** The holistic system design approach allows us to experience with and derive novel reliability solutions. In particular, the intelligent diagnosis algorithm is made possible because the checkpoint/replay recovery mechanism is available. In another example, heuristic detection mechanisms that can cause false-positive detections can be used in the SWAT detection module because the SWAT firmware has the capability to determine, through diagnosis, whether a false-positive detection has occurred at runtime. This effectively improves the coverage and latency of the SWAT detection module.
- **Amortizing overhead across other system functions.** Our view of monitoring for software symptoms of hardware bugs is inspired by work on on-line software bug detection [21, 24, 41, 86, 87, 88]. Our approach can leverage software bug detection techniques for hardware fault detection and vice versa, amortizing overheads for different system functions. In the long term, we believe the SWAT approach can bring hardware and software reliability solutions together and evolve into a unified frame-

work that tackles both types of reliability threats, essentially treating them a single system reliability challenge.

3.4 Summary

This chapter gives a brief overview of the inner workings of the SWAT system. Specifically, we have described how SWAT detects, diagnoses, recovers, and/or repairs a faulty hardware component. By focusing on faults that are harmful to the software, SWAT can leverage very low cost and effective symptom monitors to make up the efficient always-on detection scheme. Because the cost of the detection module is minimized, the common case operation is essentially optimized, resulting in a very low cost reliability solution.

While SWAT's detection scheme is low cost, it is not without tradeoffs. In particular, as multiple types of errors can all manifest as symptoms, the diagnosis process is relatively complex. Nevertheless, we are willing to have a more complicated and potentially expensive diagnosis mechanism in SWAT because diagnosis is a rare case operation and hence does not have a significant impact on the overall system cost.

Error recovery, on the other hand, is an interesting module as it is invoked in both fault-free and faulty operations. Obviously, the fault-free overhead of the recovery scheme has to be kept low in order for SWAT to be deployable to the masses.

Overall, in this chapter, we have given a high level view of the SWAT system. In the rest of this thesis, we first present the detection module in Chapter 4. Then, we describe the diagnosis scheme in detail in Chapter 5. After that, the recovery module is explored in Chapter 6.

Chapter 4

SWAT Detection

As the error detection mechanism needs to be always on in all fault-tolerant systems, its cost has a huge impact on the overall system cost. In order to provide a low-cost reliability solution, minimizing the overheads incurred by the detection mechanism is a must.

Taking this fact into account, we make a key observation that a hardware fault is only considered harmful if it affects software execution. Hence, hardware error detection mechanisms only need to handle hardware errors that propagate through high levels of the system and become observable to the software. In other words, one can detect hardware errors after they propagate into the software and appear as software bugs.

The SWAT error detection module follows exactly these observations and employs a suite of monitors of anomalous software behavior (called symptom) for hardware error detection. By using this approach, the error detection module achieves minimal cost in two ways. First, the symptom detectors, by nature, handle all hardware faults that matter and ignore those that do not. This greatly reduces excessive overhead spent on handling faults that would have been masked at various hardware and software levels. Second, because the symptom monitors themselves can be designed to catch simple software misbehavior that are easy to detect, they can be implemented at extremely low cost.

In this chapter, we start with very simple detectors that can be realized with minimal hardware cost and no software support [36]. Then, we introduce a software-assisted detector that leverages a well-known software bug detection technique for detecting hardware faults [70].

4.1 Hardware-Only Software Anomaly Monitors

To minimize the cost of the error detection module, we started with symptom monitors that can be implemented with near-zero hardware overhead. In an extreme case, there is one detector class that incurs zero

hardware cost. In the following, we discuss these detectors in more detail.

4.1.1 Fatal Traps

An easily detectable abnormal behavior due to a hardware fault is a *fatal* hardware trap in either the application or the operating system. A fatal trap is typically not thrown during a correct program execution. In a SWAT system running Solaris, the following traps are denoted as fatal traps – RED (Recover Error and Debug) State Trap (thrown when there are too many nested traps), Data Access Exception Trap, Division By Zero Trap, Illegal Instruction Trap, Memory Misaligned Trap, and Watchdog Reset Trap (thrown when no instruction retires in the last 2^{16} ticks). Using these traps as symptoms of hardware faults requires no additional hardware overhead and such a trap would simply invoke the SWAT firmware that performs further diagnosis and recovery as needed (Chapter 3).

4.1.2 Hangs

Another possible abnormal behavior due to a fault is a hang in the application or OS. Previous work has proposed hardware support to detect hangs with high fidelity, but with some area and power overhead [51]. Several optimizations to that work are possible. For example, a detector based on a heuristic can initially be used (e.g., based on the frequency of branches) – if that heuristic is satisfied, then a more complex mechanism involving hardware or software can be invoked. We, however, choose to implement a detector with a simpler heuristic to lower the overheads. In particular, we developed a heuristic based on monitoring all executed branches and detecting tight loops that have a large number of iterations by employing a table of counters. Figure 4.1 shows this table in greater detail. Each entry of the table consists of three fields: a partial tag of the PC, an instruction count that identifies when the last instance of the branch instruction retired, and a loop counter. The tag is for distinguishing among different branches that have the same index. The instruction count records when the last branch instruction retired and contains the value from the performance counter that tracks retiring instructions (already available in modern processors). The instruction count field helps the hang detector determine the size of the loop, in number of instructions. The loop counter keeps track of the number of iterations of the current invocation of the loop.

With this table, the hang detector operates as follows. Whenever a branch instruction retires, a part of

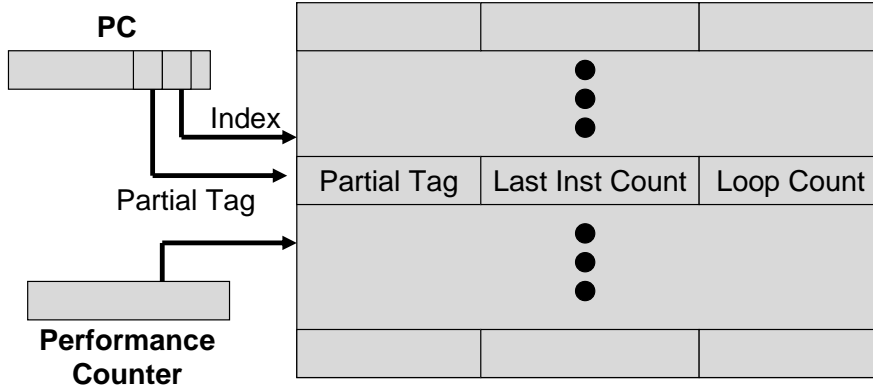


Figure 4.1: Hardware structure used for hang detection.

its PC is used to index into the table to look up the tag. If the tag does not match the current branch, a new loop is identified and the following actions are taken: (1) the tag is updated with the tag bits of the current branch's PC, (2) the instruction count field is updated with the current count from the performance counter, and (3) the loop counter is reset to zero. If a tag match occurs, then a dynamic branch with the same PC has retired previously and this branch may be part of the identified loop. To identify whether this is an iteration of a tight loop, the distance from the last invocation of the branch is computed by subtracting the current instruction count (from the performance counter) with the one in the entry. If the distance exceeds a preset range threshold, this is assumed not to be a tight loop. Since the current branch may be the beginning of a new tight loop, the instruction count field is updated and the loop counter is reset to zero. If the distance is within the range threshold, a tight loop is identified and the loop counter is incremented. If the loop counter exceeds a pre-defined iteration threshold, a potential hang is detected.

We determined the size of the table empirically from our experiments and found that a 128-entry table with each entry containing an 8-bit tag, an 8-bit instruction count, and an 18-bit loop count is sufficient. This hang detector, consisting of a table of counters, a 64-bit range threshold register, and an 18-bit iteration threshold register, consumes a total of 555 bytes. While this table is already reasonably small, further optimization in size (e.g., using branch frequency to filter out branches that are not executed frequently, allowing the use of a smaller table) is possible but we did not explore further. As this is a heuristic detector, false positives are possible but they can be identified and properly handled in SWAT. Section 4.3 discusses how SWAT handles a false-positive detection.

4.1.3 High-OS Activity

As typical OS invocations, due to traps or interrupts, take 10s or 100s of instructions, an execution that spends an excessive amount of time in the OS, without returning to the application, is a potential symptom of a fault. Such a detector can be implemented using a simple performance counter (already provided in modern processors) that counts the number of contiguous privileged instructions and invokes the SWAT firmware if a preset threshold is exceeded. We call this the High-OS detector.

However, we found exceptions to this observation. First, on a timer interrupt after the allocated time quantum for the application expires, the OS scheduler may execute for much longer. Second, for system calls (e.g., I/O), we observed that the OS may execute for much longer (10^5 or 10^6 instructions) before returning to the application. Third, there are applications (e.g., servers implemented as daemons) that voluntarily go to sleep and let the OS take over; without another runnable process, the execution stays in the OS. For these cases, we disable the High-OS detector to avoid false positive detections. Other false positives, nevertheless, are possible. Section 4.3 discusses how these cases are handled.

4.1.4 Kernel Panic

To ensure system integrity, modern operating systems are developed with a wide variety of error checking mechanisms to contain many different errors. While some violations of these checks are recoverable by the OS, some of them are critical and can cause the OS kernel to panic. A kernel panic, therefore, indicates a system anomaly. To monitor this symptom, a debug register can be used to watch for whether the panic function in the OS is executed. As modern processors already provide this register for software debugging, there is no additional hardware cost for detecting this symptom. However, OS support is needed to identify the kernel panic function.

Similar to High-OS, the Panic detector aims to detect anomalous behavior in the OS. However, while a High-OS detection may occur in normal software execution due to possible false positives, a Panic is never thrown by the OS during normal execution.

4.2 Software-Assisted Software Anomaly Monitors

Since SWAT focuses on hardware faults that propagate to the software and eventually appear as software bugs, we can potentially leverage existing software bug detection techniques to detect hardware faults. As a first step in this approach, we investigate detectors that are based on program invariants, a well-known method for detecting software bugs [21].

4.2.1 Range-Based Likely Program Invariants

A *program invariant* at a particular program point P is a property that is guaranteed to hold at P on all executions of the program. Static analysis is the most common method to extract such sound invariants. A combination of offline invariant extraction pass and static analysis, or theorem proving techniques, has also been suggested to extract sound invariants [53]. However, current techniques are not scalable enough to generate sound invariants for real programs. Also, they cannot identify algorithm-specific properties that are not explicit in the code (e.g., some inputs are always positive).

Likely program invariants are properties involving program values that hold on many executions on all observed inputs and are expected to hold on other inputs. Extracting likely program invariants is easier than extracting sound invariants as we do not need expensive static analysis methods to prove program properties and can identify algorithm specific properties. The extraction can be done either online or offline. In the online methods, invariants are extracted and used during program execution in the production runs. Online extraction, however, can present unacceptable overheads to program execution, and may in fact be infeasible without hardware support. The offline approach, on the other hand, extracts invariants in a separate pass during program testing or debugging, and these generated invariants can be used later during the production runs. During the testing phases of software development, the extra overhead of invariants extraction can be tolerated. This makes offline invariant extraction a powerful method, allowing the use of more complex invariant mining techniques that would not be feasible in the online methods. With compiler support, this “training” phase can be done transparently at development time.

Since likely invariants are unsound invariants, they may not hold on some inputs. Therefore, during production runs, false positives can occur. In the presence of permanent faults, SWAT must be able to correctly tell apart a false positive invariant violation and an invariant detection caused by a permanent

hardware fault. We discuss how false positives are handled in Section 4.3.

While there are many types of likely invariants, we can broadly classify them into three categories. *Value based* invariants specify properties involving only program values, and can be used for a variety of tasks including software bug detection, program understanding and refactoring, etc. [20, 39, 31, 40, 25]. *Control flow based* invariants specify properties of the control flow of the program, and have been used previously to detect control-flow errors due to transient faults [79, 78, 22]. *Program counter based* invariants specify program properties involving program counter values, and have been proposed for detecting memory errors in programs during debugging [85].

The control flow based and program counter based invariants can detect control flow or memory access errors, which generally result in anomalous software behavior that can be detected by the hardware-only detectors in SWAT. For example, an erroneous control flow can result in a *fatal trap*. In contrast, fault-induced deviations in values that do not cause control flow or memory access errors are more difficult to detect with hardware-only detectors and may result in incorrect program outputs. We believe that value based invariants are effective for detecting these errors that only corrupt data, and explore the use of value-based invariants to detect permanent faults.

As a first step towards using likely program invariants for permanent hardware faults, we use a particular form of value-based invariants known as range-based invariants. A range-based invariant on a program variable x will be of the form $[MIN, MAX]$, where MIN and MAX are constants inferred from offline training such that $MIN \leq x \leq MAX$ is true for all the training runs.

These range-based invariants are suitable for error detection for various reasons. These types of invariants can be easily and efficiently generated by monitoring program values. They are also composable – the invariants can be generated for each training input separately and can then be combined together to generate invariants for the complete training set. These invariants are also much easier to enforce within the checking code compared to other forms of invariants as they are simple and involve a single data value.

The following describes the steps we take to generate invariants for detecting hardware faults. We use the LLVM compiler framework for these steps.

Invariant Generation

The likely range-based invariants are derived by training the targeted applications with a variety of inputs. The data range of each static store instruction is collected for each training input. We decided to monitor only the store values as checking values stored to memory has the most potential to catch faults, as all necessary computations eventually pass their results to stores. Also, monitoring only the stores helps us keep the overhead of detection low. We monitor stored values of all integer types (both signed and unsigned) of size 2, 4, and 8 bytes as well as single and double precision floating point types. We do not monitor integer stores of size 1 byte (character data types), as they represent only a small range of values and hence may be ineffective to detect faults.

Invariant Insertion

The invariants generated from the previous phase then need to be inserted into the code to check the values being stored for hardware fault detection. To accomplish this, we take the generated invariant ranges and then insert calls to the invariant checking code at the LLVM byte-code level through an instrumentation pass in the compiler. At this point, the resulting application binary contains checking code that is capable of detecting invariant violations.

4.3 Handling False Positives

After a symptom is detected, if the diagnosis (described in Chapter 3) determines that the symptom was not caused by a hardware fault, this symptom is deemed a false positive for the presence of a hardware fault. In these cases, fatal traps and kernel panics are essentially symptoms of software bugs and will simply be propagated to the appropriate software layer as usual. The additional diagnosis latency in these cases is acceptable since it is incurred in the case of a fault, albeit in software.

For symptoms such as hangs and high OS activity, the detection mechanisms themselves are prone to false positives as they are based on heuristics. When the diagnosis determines that one of these symptoms is a false positive for the presence of a hardware fault, the execution will simply continue. In this case, the diagnosis latency is an overhead for fault-free execution. To evade this overhead in the future, the SWAT

firmware can increase the thresholds in these detectors to avoid a similar kind of false positive. However, because higher thresholds potentially lead to higher detection latencies, the SWAT firmware can periodically decrease the threshold if no false positives are detected. In general, there is a tradeoff between the latencies of these symptom detectors and their false positive rates.

Likely invariants are also prone to false positives because they are expected to hold on most, but not all, program inputs. After detecting invariant violations, we use the SWAT diagnosis module to identify false positives. A diagnosed false positive means that the particular program input leads to data values that were never seen during the training runs. Hence, the false positive detection is likely to occur again in subsequent execution. Because all SWAT detections invoke the diagnosis module which can be expensive, false positives can therefore be costly. To limit the overhead incurred by false-positive invariant violation detections, SWAT diagnosis disables the offending invariant check.

4.4 Methodology – Base Environment

The main goal of our experiments is to study the effectiveness of the SWAT detection, diagnosis, and recovery components when the system has a fault. To achieve this, we conduct fault injection experiments on a common base environment. In this section, we describe the base simulation environment in Section 4.4.1 and the fault models used in Section 4.4.2.

4.4.1 Base Simulation Environment

Ideally, to evaluate the detection, diagnosis, and recovery components of SWAT, we would like to inject hardware faults into a real system or a low-level (e.g., gate level) simulator. However, modern processors do not provide enough observability and controllability to perform the microarchitecture-level fault injections that are of interest to us. We therefore use simulation. Although low-level simulators would provide the ability to use more accurate fault models, they present a trade-off in speed and the ability to model long running workloads with OS activity. Since we need to evaluate the impact of persistent faults on the software and need to simulate for long periods (millions of cycles), gate-level simulation was not feasible. We therefore conduct our fault injection campaign in a microarchitecture-level simulator. (Chapter 7 presents a new efficient way for simulating gate-level faults with microarchitecture-level simulation speeds.)

Base Processor Parameters	
Fetch/Decode/Execute/Retire rate	4/cycle
Functional units	2 Int add/mul, 1 Int div, 2 Load, 2 Store, 1 Branch 2 FP add, 1 FP mul, 1 FP div/Sqrt
Integer FU latencies	1 add, 4 mul, 24 divide
FP FU latencies	4 default, 7 mul, 12 divide
Reorder buffer size	128
Register file size	256 integer, 256 FP
Unified Load-Store Queue Size	64 entries
Base Memory Hierarchy Parameters	
Data L1/Instruction L1	16KB each
L1 hit latency	1 cycle
L2 (Unified)	1MB
L2 hit/miss latency	6/80 cycles

Table 4.1: Parameters of the simulated processor.

To simulate a system with faults, we use a full system simulator comprising the Wisconsin GEMS microarchitectural and memory timing simulators [42] in conjunction with the Virtutech Simics full system simulator [80].

Together, these simulators provide cycle-by-cycle microarchitecture level timing simulation of a real workload running on a real operating system on a modern out-of-order superscalar processor and memory hierarchy (Table 4.1). In particular, we simulated six SpecINT2000 (bzip2, gcc, gzip, mcf, parser, and twolf) and 4 SpecFP2000 (ammp, art, equake, and mesa) applications on Sun Solaris 9 running SPARC V9 ISA (Figure 4.2(a) shows the simulation environment). We also simulated two server applications, Apache web server and SSH daemon (described in Table 4.2), on OpenSolaris. Since server workloads are driven by requests made by client systems, we created an environment in Simics that consists of two separate systems connected by a simulated network (shown in Figure 4.2(b)).

To inject faults, we leverage the timing-first approach [44] used in the GEMS+Simics infrastructure. In this approach, an instruction is first executed by the cycle-accurate GEMS timing simulator. On retirement, the Simics functional simulator is invoked to execute the same instruction again and to compare the full architecture state in GEMS and Simics. This comparison allows GEMS the flexibility to not fully implement a small (complex and infrequent) subset of the SPARC ISA – GEMS uses the comparison to make its state consistent with that of Simics in case of a mismatch that would occur with such an instruction.

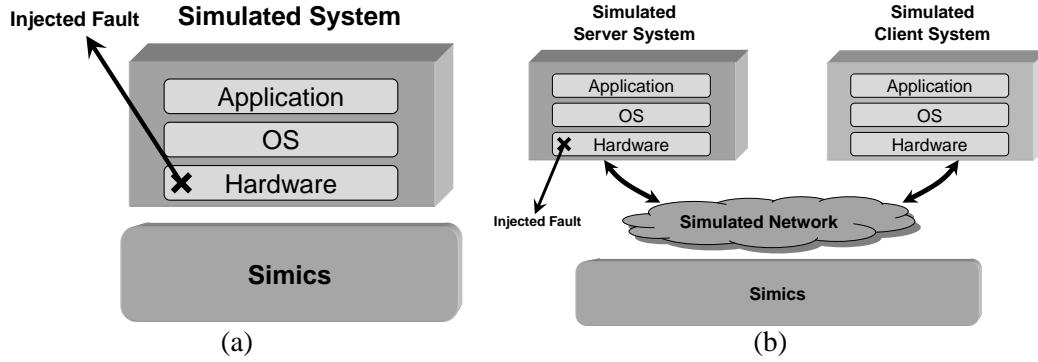


Figure 4.2: Simulation environment. (a) A single-system environment that runs SPEC applications on a commercial OS. (b) A two-system environment that runs server applications on one system and client applications on another system.

We modified this checking mechanism for the purposes of microarchitectural fault injection. We inject a fault into the timing simulator’s microarchitectural state and track its propagation as the faulty values are read through the system. When a mismatch in the *architectural state* of the functional and the timing simulator is detected, we check if it is due to the injected fault. If not, we read in the value from Simics to correct GEMS’ architectural state. However, if the mismatch is because of an injected fault, we corrupt the corresponding state in Simics (register and memory) with the faulty state from GEMS, ensuring that Simics continues to follow GEMS’ execution trace, upholding the timing-first paradigm.

We say an injected fault is *activated* when it results in corrupting the architectural state, as above. If the fault is never activated, we say the fault is *architecturally masked* (e.g., a stuck-at-0 fault in a bit that is already 0 or a fault in a misspeculated instruction are trivially masked). Since we know the privilege mode of the retiring instruction that corrupts the state, we can determine if a fault leads to any corruption in the architectural state of the OS or the application. As discussed later, this information has important implications for recovery.

Although in the fault-free executions of the SPEC workloads, the simulated applications are not OS-intensive ($< 1\%$ OS activity in our simulated window), we show later that fault injection significantly increases OS activity. For server workloads, as the OS activity is already high (50+% in our simulated window), we observe later that the injected faults are highly likely to corrupt the OS. Because hardware faults can corrupt the OS state for our workloads, it is critical to model the OS and its interaction with the

Benchmark	Description	Fault-free Output
Apache	Provides webpages and files at a website to requesting clients through the HTTP protocol. Four worker threads listen to incoming requests from a synthetic driver with 20 threads, obtained from the cURL [14] utility.	Each client thread receives the assigned files that are the same as stored on the server.
SSH Daemon	Provides files to the clients using the SSH protocol. One daemon thread listens to a synthetic client system with 8 threads, and spawns threads with added connections.	Each client thread receives the assigned files that are the same as stored on the server.

Table 4.2: Description of server workloads.

μ arch structure	Fault location
Instruction decoder	Input latch of one of the decoders
Integer ALU	Output latch of one of the Int ALUs
Register bus	Bus on the write port to the Int reg file
Physical integer reg file	A physical reg in the Int reg file
Reorder buffer (ROB)	Source/destination register number of instructions in ROB entry
Register alias table (RAT)	Logical \rightarrow physical map of a logical register
Address gen unit (AGEN)	Virtual address generated by the unit
FP ALU	Output latch of one of the FP ALUs

Table 4.3: Microarchitectural structures in which faults are injected. In each run, either a stuck-at fault is injected in a random bit or a bridging fault is injected in a pair of adjacent bits in the given structure.

applications in our simulations. Hence, our experiments are run in a full-system simulation environment.

4.4.2 Fault Models

As phenomena such as wear-out or infant mortality due to incomplete burn-in [8, 9, 84] become increasingly important, we would want to model them at the microarchitecture level. However, since precise fault models for wear-out are still a subject of research [73] and we do not have access to all gate-level modules of a superscalar processor, we use the well established stuck-at-0 and stuck-at-1 fault models as well as the dominant-0 and dominant-1 bridging fault models injected at the microarchitecture level. While the stuck-at fault models apply to faults that affect a single bit, the bridging fault models concern faults that affect adjacent bits. The dominant-0 bridging fault acts like a logical-AND between the adjacent bits that are marked faulty, while the dominant-1 bridging fault acts like a logical-OR. Table 4.3 lists the microarchitectural structures

and locations where we inject faults.

Fault Injections for SPEC Workloads

For each structure, we inject a fault in each of 40 random points in each application (after initialization), one injection per simulation run. For each application injection point, we perform an injection for each of the 4 fault models (two stuck-at and two bridging). The injections are performed in a randomly chosen bit in the given structure for stuck-at faults. For bridging faults, a randomly chosen pair of adjacent bits are injected. This gives a total of 1600 fault injection simulation runs per microarchitectural structure ($10 \text{ applications} \times 40 \text{ points per application} \times 4 \text{ fault models}$) and 12,800 total injections across all 8 structures. This gives us an overall error of 0.4% at a 95% confidence, making our results statistically significant.

We also performed a total of 6400 transient fault injections (single bit flips) in the same microarchitectural structures. (The number of injections is fewer than for permanent faults because of fewer fault models.) The error is a low 0.6%, at a 95% confidence.

Fault Injections for Server Workloads

For server workloads, we focus on stuck-at faults because they are well-known standard fault models. Also, since profiling our server applications shows that the FPU was never used, we focus on the other 7 microarchitectural structures. Further, because there are only 2 server applications, we deliberately increase the number of injected faults to achieve statistical significance.

In each run, a stuck-at-0, a stuck-at-1, or a transient fault is injected in a randomly chosen bit in one of the 7 structures (all except FP ALU) listed in Table 4.3. For each of Apache and SSH daemon, we pick 4 base injection points (or *phases*) spaced sufficiently apart in the execution of the application, to capture different behaviors of the application. In each phase, for each structure, we pick 40 spatially and temporally random injection points for each of the stuck-at-0 and stuck-at-1 faults, and 80 spatially and temporally random injection points for transients (e.g., 40 different physical registers, each with stuck-at-0, and stuck-at-1 faults, and 80 different RAT entries, each with a transient fault in a random bit). This gives us a total of 4480 permanent faults ($2 \text{ applications} \times 4 \text{ phases} \times 7 \text{ structures} \times 40 \text{ random points} \times 2 \text{ fault models}$) and 4480 transient faults (same as the above, except with 80 random points, and one fault model). This gives a

low overall error of 0.2% at a 95% confidence, making our results significant.

We inject the fault at the server system only and attempt to detect and recover faults at the server without involving the client.

While the experiments for the SWAT detection, diagnosis, and recovery schemes are based on this environment in general, there are a few exceptions (e.g., software-assisted invariant detection). In the rest of the thesis, we explicitly point out the exceptions.

4.5 Methodology – Detection

This section focuses on the environment used in SWAT detection experiments. In Section 4.5.1, we first describe the parameters used for the SWAT symptom detectors. We then present our experimental setup for capturing the impact of each injected fault in Section 4.5.2. In Section 4.5.3, we describe the metrics used in the experiments.

4.5.1 Symptoms Studied

We employ the monitors described in Sections 4.1 and 4.2 to detect the injected faults. For the software-assisted invariant detectors, due to the lack of realistic program inputs for some of the applications, we applied the invariant detectors to five SPEC applications. Since previous work has investigated application-specific detectors for transient faults [54], our experiments for the software-assisted invariant detection focus on permanent faults. In the following, the parameters of each of the symptom monitors are discussed.

- **Fatal Traps.** The following fatal traps are monitored – RED (Recover Error and Debug) state trap (thrown when there are too many nested traps), Data Access Exception trap, Division by zero trap, Illegal instruction trap, Memory misaligned trap, and Watchdog reset trap (thrown when no instruction retires in the last 2^{16} ticks). Further, in our experiment, a fault can cause a fatal trap in either the application or the OS.
- **Hangs.** For our experiments, we set the range threshold to be 200. For the iteration threshold, we use 100,000 for SPEC workloads and 250,000 for server workloads. We identified the iteration thresholds through profiling the fault-free executions of the applications. We consider both hangs in the

application and the OS.

- **High-OS activity.** We look for a threshold of over 30,000 contiguous OS instructions, *excluding* cases where the OS is invoked via a system call, a timer interrupt, or an idle loop. This threshold corresponds to a conservative latency which is 3 times the maximum observed scheduler latency. We switch off High-OS for the server workloads. While we can tune High-OS for the server applications, their inherently high OS activities make it difficult to do so. For example, since the server daemon may be blocked (put to sleep) in different parts of the OS, the High-OS detector needs to be tuned to ignore these regions of code to prevent false positive detections. We employ kernel panic (described below) in place of High-OS for server workloads.
- **Kernel Panic.** We watch for 11 OS (privileged) instruction addresses of the kernel panic functions in OpenSolaris to detect when a panic is thrown. We switch on this symptom in place of High-OS for the server workloads because of the reason given above. From our experiments, we found that kernel panic is adequate for detecting many faults in server workloads. We did not use this symptom for SPEC workloads as we later show that the High-OS detector is sufficient to detect a large fraction of faults.
- **Range-Based Likely Invariants.** Using the LLVM compiler framework, we generate the invariants for and insert the invariant checking code into five SpecCPU 2000 benchmarks – four SpecInt benchmarks (gzip, bzip2, mcf, parser) and one SpecFP benchmark (art). The “test” and “train” input sets formed part of our training set. Different techniques were used to generate more inputs depending on the benchmarks. For three benchmarks (gzip, bzip2 and parser), we collected random inputs from external sources. For mcf, a script was used to generate random inputs, while for art, different input options were used to generate invariants. We did not use other SPEC benchmarks because of various training input collection and compilation issues. Nevertheless, obtaining inputs will not be a problem in practice as developers test their programs on many inputs during the testing phase. Invariant generation and insertion can be easily done during testing through a compile-time pass.

After the application binaries are instrumented with invariant checking code, we inject faults while the applications are running the “ref” inputs (not part of the training input set).

4.5.2 Fault Simulation Experimental Setup

The main objective of our fault injection experiment is to investigate the effectiveness of SWAT’s symptom detectors in detecting the underlying hardware faults. To achieve this, each fault is injected into the designated microarchitectural structure and simulated in the microarchitectural simulator for certain number of instructions to see if it leads to any symptom. If no symptom is detected in this detailed simulation, we investigate the impact of the fault on the software by functionally simulating the application to completion. The following presents the detailed experimental setup.

Microarchitectural Simulation

During this detailed simulation, after a fault is injected, the simulation runs for at least 10 million instructions. If an injected fault results in a monitored symptom, it is considered *detected and recoverable*. Hence, this detection is counted towards part of the coverage and its latency is measured (described in Section 4.5.3). If a fault never corrupts the architectural state 10 million instructions after the injection, it is considered *architecturally masked*. If the fault corrupts the architectural state but does not result in a symptom after simulating for 10 million instructions since the corruption, we proceed to functional simulation to investigate its impact on the system.

Functional Simulation

A fault that corrupts the architectural state may or may not have an adverse effect on the system. To determine the eventual effect of an injected fault, we simulate the application to completion using functional simulation. We did not use microarchitectural simulation because it would take too long. Because of the lack of microarchitecture-level details during functional simulation, fault activation will not occur in this phase and the injected fault appears as an intermittent fault.

During the functional simulation phase, there are three possible outcomes. If a fault causes the application or the system to crash (e.g., panic) or hang (not responsive), we classify this case as a *detected unrecoverable error (DUE)*. Since we do not know the latencies and they may (or may not) be too long for recovery, we conservatively consider these faults as unrecoverable and do not count them towards the detection coverage of SWAT. If the application finishes execution normally and the resulting output is identical

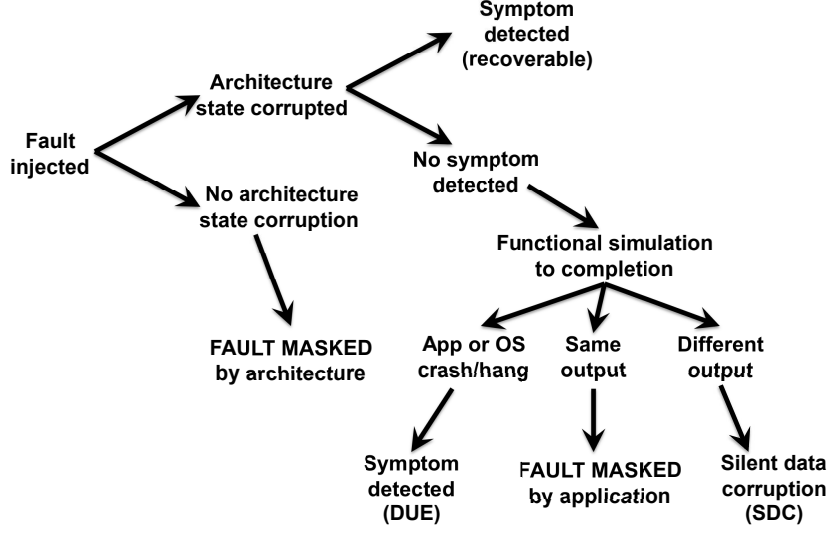


Figure 4.3: Outcomes of an injected fault. If the injected fault is not detected within 10M instructions, the fault is removed (no new fault activation, but software state may already be corrupted at this point) and the application is functionally simulated to completion to identify its effect on the application’s outputs or whether it causes a detected unrecoverable error (DUE).

to that of the fault-free execution, the injected fault is considered to be *masked by the application*. In this case, SWAT correctly ignores this benign fault, avoiding the potential overheads of diagnosis and recovery. However, if the resulting output is different from the one produced by the fault-free execution, we categorize this case as a *silent data corruption (SDC)*.

In Figure 4.3, we show all the possible outcomes of an experiment as described above.

4.5.3 Metrics

The effectiveness of a detection mechanism is typically determined by *whether* an injected fault is detected and *how long* the detection mechanism takes to detect the fault. Hence, we focus on detection coverage, rate of silent data corruptions, and detection latency in our experiments.

Coverage: The coverage of a detection mechanism is the percentage of unmasked faults it detects. While detecting a fault is essential, the eventual system reliability depends on whether the detected errors are recoverable. Here, we focus on the detections that occur within 10 million instructions as they are believed to be recoverable with existing hardware checkpointing schemes [59, 74] (we take a closer look at actual

system recoverability in Chapter 6). Hence, we define coverage as follows.

$$Coverage = \frac{\text{Number of faults detected within 10M instructions}}{\text{Number of faults injected} - \text{Number of masked faults}}$$

where the masked faults are ones that are masked by either the architecture or the application.

SDC rate: The rate of silent data corruption is defined as the percentage of injected faults resulting in silent data corruptions.

Detection latency: We report fault detection latency as the total number of instructions retired from the first architecture state corruption (of either OS or application) until the fault is detected. For detections where the faults do not corrupt the architecture state, we consider them to have latencies of zero instructions.

For our software-assisted invariant detectors, we measure the above metrics the same way as hardware-only detectors. Since likely invariants may result in false positive detections, we look at the false positive detection rate when we vary the number of training inputs. High false positive rates would result in frequent invocations of the diagnosis routine, incurring significant performance overhead during normal (fault-free) execution. As the invariant checking code inserted into the application binaries incurs performance overhead all the time, this overhead is also measured in real systems. If the detectors impact performance substantially, the overall cost of the SWAT system would be too expensive for commodity systems. In the following, we define the false positive rates and overhead of our invariant detectors.

False positive rate: The false positive rate of the likely invariant detection mechanism for a particular application is the percentage of all static invariants in the application binary that trigger a false positive detection. (Once a static invariant triggers a false positive, it is deactivated by the diagnosis routine.)

Overhead: The performance overhead of the invariant detection mechanism is calculated as follows.

$$Overhead = \left(\frac{\text{Execution time of application enhanced with invariant checking}}{\text{Execution time of original application}} - 1 \right) \times 100\%$$

4.6 Results – Hardware-Only Detectors

The bulk of our experiments here focus on permanent hardware faults (vs. transients) because of the increasing importance of such faults due to phenomena such as wear-out and insufficient burn-in [8], because transients have already been the subject of much recent study, and because permanent faults pose significant challenges different from transients. For example, a permanent fault may manifest to software faster than a transient (because it lasts longer), but for the same reason, it is less likely to be masked and more likely to corrupt the OS with an irrecoverable system failure (unless intercepted quickly). Further, after a permanent fault is exposed, the system must diagnose its source and repair or reconfigure around the faulty unit. This is generally expensive, limiting the number of affordable false positives (unlike some detection techniques for transients [81]). Nevertheless, we summarize the main experimental results of the hardware-only detectors for transients.

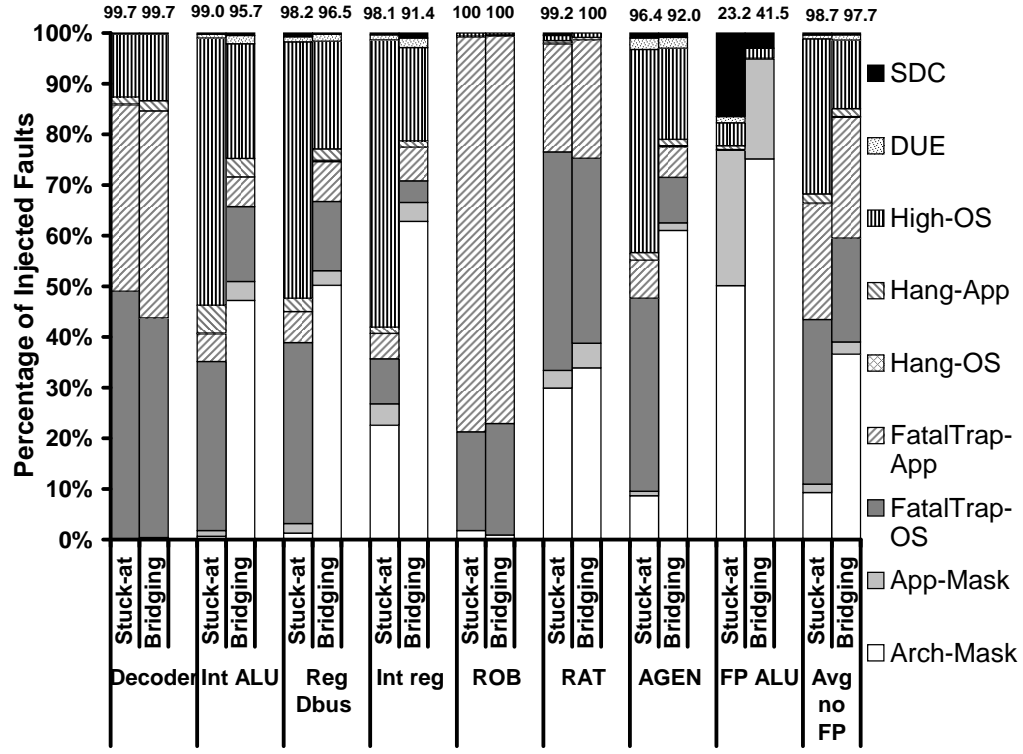
In the following, we discuss the experimental results of hardware-only detectors while Section 4.7 presents those of software-assisted detectors.

4.6.1 Detection Coverage

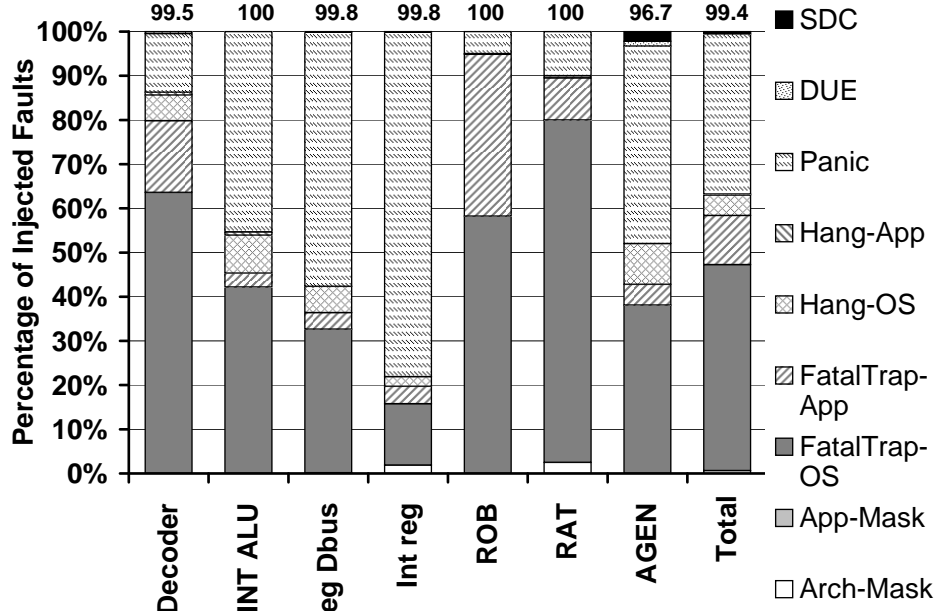
Figures 4.4(a) and (b) show the outcomes of our permanent fault injection campaign using SWAT’s hardware-only detectors on SPEC workloads and server workloads, respectively. We categorize our faults as architecturally masked (*Arch-Mask*), application masked (*App-Mask*), detected within 10M instructions and recoverable (*Fatal-Trap*, *Hang*, *High-OS*, *Panic*) in either the application or the OS (*App* or *OS*), detected but not recoverable (as detection latency is more than 10 million instructions) (*DUE*), and Silent Data Corruptions (*SDC*). The number on top of each bar shows the *Coverage* for the particular microarchitectural structure.

The key high-level results are:

- For the cases studied, permanent faults in most structures of the processor are highly software visible. 98% of faults that are not masked (except for the FPU) are detected for SPEC workloads while 99% of unmasked faults are detected for server workloads using our simple detection mechanisms. This clearly demonstrates the effectiveness of using high-level software symptoms to detect permanent hardware faults.



(a)



(b)

Figure 4.4: Coverage of SWAT hardware-only detectors for (a) SPEC workloads for both stuck-at and bridging permanent faults and (b) server workloads for stuck-at permanent faults.

- For the FPU, 73% of the activated faults are not detected, suggesting that alternate techniques may be needed (e.g., redundancy in space, time, or information) for the FPU.
- For SPEC workloads, many of the faults are detected when running the OS code (the FatalTrap-OS, Hang-OS, and High-OS categories), even though the fault-free applications themselves are not OS intensive. An even greater fraction of the faults are detected when running the OS code for server workloads, which have higher OS activity than SPEC.
- For SPEC workloads, the FatalTrap and High-OS categories make up the majority of the detections (68% and 30% respectively of all detected faults) while the Hang is the smallest (only 2.3%). For server workloads, FatalTrap and Panic detectors make up 58% and 37% of all detections, respectively. Only 5% of the detections are Hangs.
- Only 0.3% of the injected faults result in silent data corruptions of SPEC applications and 0.3% of injections corrupt the server applications silently without being detected. The rest eventually lead to application/OS crashes/hangs or are masked by the application.

The rest of this section provides a deeper analysis to understand the above results.

Analysis of Masked Faults

For stuck-at faults injected in both SPEC and server workloads, Figures 4.4(a) and (b) show low architectural masking rates for many structures. This is because the injected fault is a permanent fault that potentially affects every instruction that uses these faulty structures during its execution. Server workloads, in comparison, have significantly lower masking rates than SPEC workloads. 0.6% of injections in server workloads are masked while 9.3% of injections (excluding FPU) in SPEC workloads are masked. This is because server workloads, which generally have higher OS activities, tend to utilize the microarchitectural structures more rigorously. For example, more nested function calls in server workloads cause higher utilization of the windowed architectural register file in SPARC, making RAT faults less likely to be masked.

Exceptions to low architectural masking rates are stuck-at faults injected into the integer register file, the RAT, the AGEN, and the FPU for SPEC workloads, where the architectural masking rate ranges from 8.6% in AGEN to 50% in FPU. On the other hand, for server workloads, 2% of the injected faults in the

integer register file and the RAT are masked architecturally. Architectural masking for an integer (physical) register occurs if it is not allocated in the microarchitectural simulation window of 10 million instructions. Similarly, a RAT fault is masked if it affects the physical mapping of a logical register that is not used in this window. An AGEN fault is masked if the injected bit does not change throughout the execution. The high FPU masking rate occurs because of the integer applications.

Bridging faults in SPEC workloads also see the above phenomena for architectural masking. Additionally, most structures on the 64 bit wide data path (INT ALU, register DBus, integer register file, and AGEN) see a significantly higher architectural masking rate for bridging faults than for stuck-at faults. This difference stems from faults injected in the upper 32 bits of the 64 bit fields (roughly half of total fault injections in those structures). Since many computations only use the lower 32 bits, the higher order bits are primarily sign extensions, with either all 0s (for positive numbers) or all 1s (for negative numbers). In either case, since adjacent bits are identical, bridging faults are rarely activated for higher order bits, resulting in a higher masking rate for these faults.

Relative to architectural masking, application masking in SPEC workloads is small but significant (4.7% of total injections). Many of these cases stem from faults injected in the higher order bits of the 64 bit data path – in some cases, these appear as architecture state corruptions (because the full 64 bit field is examined), but are actually masked at the application level due to smaller program level data sizes.

In contrast, there is only one case in server workloads that results in application masking. One possible reason for this low application-level masking rate is because of the generally high OS activity. Since the OS is more control-intensive than applications, an activated fault is more likely to corrupt the system state and cause a visible symptom instead of being masked by the software.

Nevertheless, these faults illustrate a benefit of our symptom-based detection approach since these benign faults are correctly ignored by our detectors.

Analysis of Detected Faults

Unmasked faults in many structures are highly visible. As these faults are permanent in nature, they are activated many times. As long as one activation affects a program path and subsequently leads to a symptom, the fault will be detected. The only undetected cases where the changes in the program paths do not lead to

symptoms are pure value corruptions. Our results, however, show that this is uncommon. In the following, we analyze the detected cases in greater detail.

1. **Large number of detections in the OS for SPEC applications.** Surprisingly, in spite of the low OS activity for the fault-free runs of the simulated benchmarks, over 65% of the detected faults are detected through symptoms from the OS (FatalTrap-OS, Hang-OS, and High-OS). Although the injected fault first corrupts the application, a common result of the fault is a memory access to an incorrectly generated virtual address. Since the address has not been accessed in the past, it invokes a TLB miss that would not have otherwise occurred. Because the SPARC TLB is software managed, this results in a trap invoking the OS. As the OS is executing on the same faulty hardware and, in general, is more control and memory intensive, the fault often will corrupt the OS state and result in a detectable symptom.

As a comparison, server workloads have more than 50% OS activity and 89% of the detections happen in the OS. In these applications, the OS is more likely to be corrupted since it is invoked more frequently. After the fault corrupts the OS execution, a symptom detection often occurs in the OS, instead of returning to the application.

2. **Fatal Hardware Traps.** 68% of the fault detections in SPEC applications and 58% of the detections in server applications are from fatal hardware traps. Figures 4.5 and 4.6 show the distribution of the different types of these fatal traps. The height of a bar is the percentage of fault injections in the corresponding structure that causes fatal traps. Fatal traps caused by the application are shown in the bottom (hatched portions) and those caused by the OS are shown on top (non-hatched portions).

An *illegal instruction* trap occurs when one or more opcode fields in an instruction is invalid. As expected, these traps result mostly for decoder faults. However, they account for <16% (<19%) of the fatal traps seen on decoder faults for SPEC (server) workloads. This is because many injected faults in the instruction word either do not affect the opcode bits, or when they do affect opcode bits, they change the instruction into another valid instruction.

The *watchdog timer reset* trap is thrown when no instruction retires for more than 2^{16} ticks. These mostly occur in the ROB and RAT, accounting for 90% and 59% of fatal traps, respectively, for SPEC

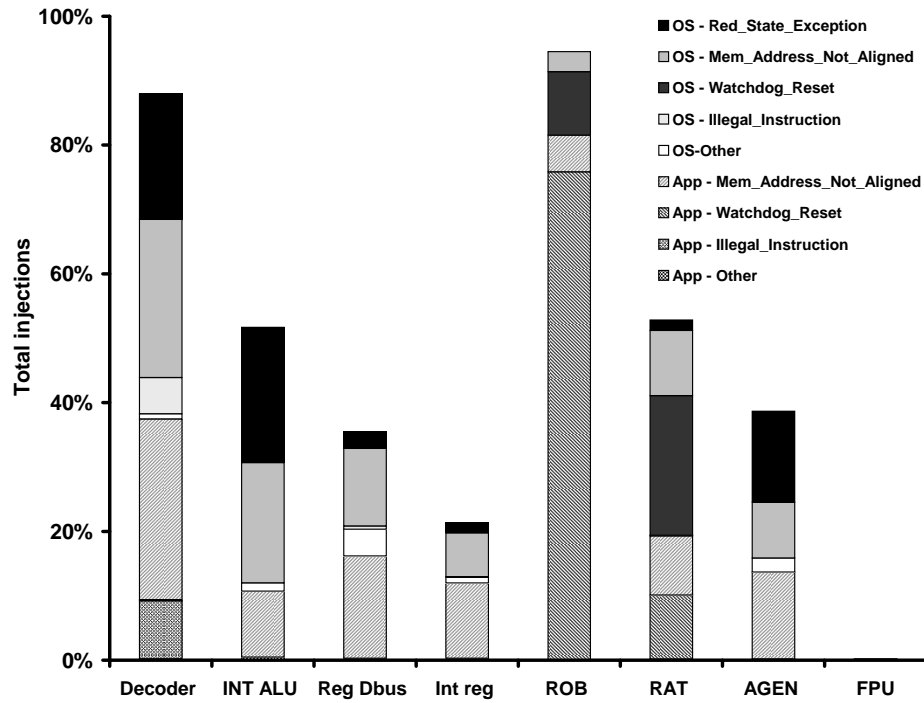


Figure 4.5: Distribution of detections by fatal traps for SPEC workloads. The *Other* category constitutes Data Access Exception, Protection Violation and Division by Zero traps, which make up <8% of detections by fatal traps. The total height of a bar is the percentage of the injected faults in the corresponding structure that caused fatal hardware traps.

workloads, and 80% and 66% of fatal traps for server workloads, respectively. Both ROB and RAT faults may change the source or destination register of an instruction. If the source is changed to a free physical register, the instruction waits for data indefinitely. If the destination is changed, the dependent instructions indefinitely wait for their source operand. For example, the corrupted logical-to-physical register mapping could result in mapping a non-free physical register (say $preg_{23}$). Now that $preg_{23}$ is mapped to two logical registers (say r_2 and r_5), any subsequent instruction that writes to r_2 (r_5) will free $preg_{23}$ and instructions that read r_5 (r_2) wait for $preg_{23}$ indefinitely (since $preg_{23}$ is freed and marked *not ready*). Since the ROB is a circular buffer and is heavily used, faults in the ROB are highly intrusive. If either one of the two source operands is mutated to point to a free register, this trap will occur. In contrast, a RAT fault induced watchdog reset trap depends on how often a particular logical register (in a large set of logical registers in the SPARC architecture) is used and hence occurs not as frequently as a ROB fault induced watchdog reset trap.

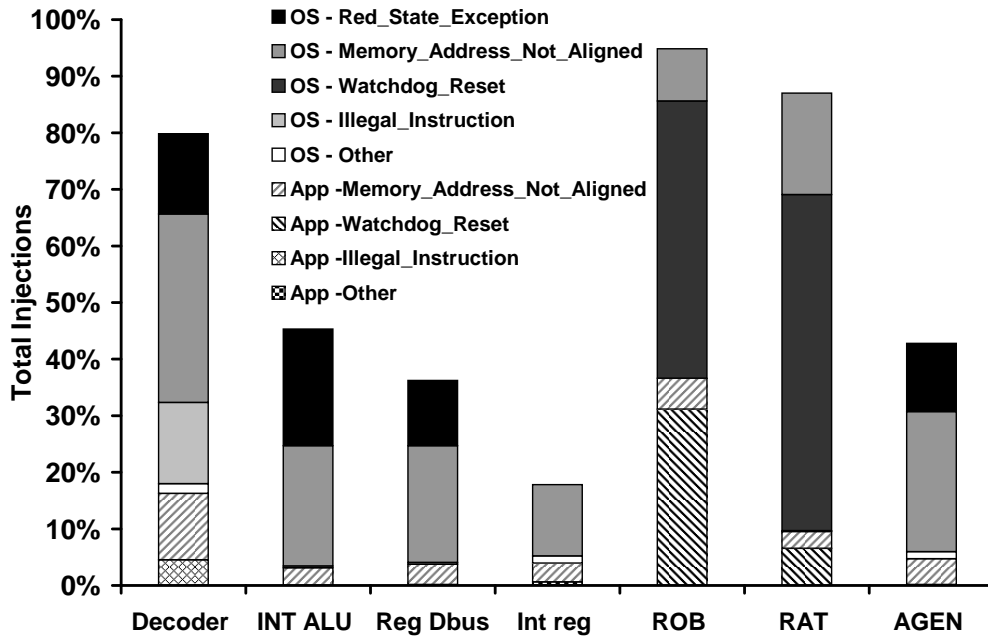


Figure 4.6: Distribution of detections by fatal traps for server workloads. The *Other* category constitutes Data Access Exception, Protection Violation and Division by Zero traps, which make up <2% of detections by fatal traps. The total height of a bar is the percentage of the total injected faults in the corresponding structure that caused fatal hardware traps.

Misaligned accesses are common in all structures, accounting for over 44% (SPEC workloads) and 43% (server workloads) of all the fatal traps thrown. Faults in most structures naturally affect the computation of memory addresses (e.g., all cases where a fault may affect the data or identity of a register used to compute an address). This often results in misaligned addresses, causing a misaligned access trap (Solaris requires addresses to be word aligned).

Red state exception is thrown when there are too many nested traps. The SPARC V9 architecture throws this exception when a trap at (maximum_trap_level - 1) occurs. The simulated processor has a maximum_trap_level of 5; i.e., at most four nested traps are allowed. This fatal trap constitutes 15% of the fatal trap detections for SPEC workloads and 14% for server workloads. For a RED state exception to occur, the injected fault first results in invoking the OS through a non-fatal trap (otherwise, the fault would have been detected). When this trap handler executes, it re-activates the fault and a nested trap results. As this fault-activating pattern repeats, a RED state exception occurs eventually.

3. **High OS.** For SPEC workloads, the High-OS symptom has the next highest detection coverage after

fatal traps (30%). In the majority of these cases, the application computes a faulty address invoking the OS on a TLB miss. The persistent hardware fault corrupts the TLB handler, resulting in the code never returning to the application.

This symptom has significant coverage overlap with fatal traps and hangs – removing this detector reduces the total coverage for all structures except FPU by about 15% (instead of the 30% if there were no overlap) for SPEC workloads. This is because most of these cases eventually also lead to fatal traps and hangs. However, even for these cases, detection using the High-OS symptom significantly brings down the detection latency (Section 4.6.3).

4. **Panics.** For server workloads, Kernel Panic detects the highest number of faults next to fatal traps and accounts for 37% of all detections. This symptom shows that modern operating systems are also very effective in catching hardware faults. As the software activates the injected fault, the OS may be invoked through a non-fatal trap. After that, the fault is activated by the OS execution and corrupts some crucial system state, causing a check of the system state to fail and resulting in a kernel panic. Hence, this result shows that the efficacy of the SWAT detection mechanism can be greatly improved if the OS can be involved for monitoring some OS-specific anomalies.
5. **Hangs.** Hangs account for less than 3% coverage for SPEC workloads and less than 5% coverage for server workloads. Comparing the two types of workloads, we find that practically all hangs occur in the application code for SPEC workloads while nearly all hangs occur in the OS for server workloads. This discrepancy is likely caused by the difference in OS activities for the two workloads. Since SPEC (server) workloads execute mostly application (OS) code, the faults are more likely to corrupt the loops in the applications (OS).

An example of a hang is when a loop index variable is computed erroneously and the loop termination condition is never satisfied. While hangs in SPEC (server) workloads may result from the OS, the High-OS (Panic) symptom catches these before the hang detector can identify them as hangs. Thus, without the High-OS or Panic detector, hangs would provide higher coverage (but at a higher latency).

Analysis of Detected Unrecoverable Errors

Faults that are unmasked and undetected within the detailed microarchitectural simulation but cause symptoms in the functional simulation have detection latencies that may or may not be short enough for full recovery (e.g., by rolling back to a software checkpoint). To be conservative, we classify these faults as detected unrecoverable errors. Nevertheless, eventual detection is better than letting faults cause silent data corruptions.

Overall, 0.5% of the detected faults result in DUEs for SPEC workloads and 0.3% of the detections are DUEs for server workloads. Across different structures, for both SPEC and server workloads, DUEs account for less than 1% of the detections for all but AGEN and FPU. For AGEN, 1.8% of the detections in SPEC applications and 1.1% of the detections in server applications result in DUE. Of the injected FPU faults that are detected while running SPEC workloads, 14% are DUEs because we have few detections in FPU to start with.

Generally, from our results, there are very few faults that are detected but unrecoverable, showing the effectiveness of the employed simple symptom monitors. In the future, the goal of the SWAT detection is to derive even better symptoms to eliminate the DUEs (i.e., detecting them at short latencies) as much as possible.

Analysis of Silent Data Corruptions

For the unmasked faults that are not detected in the microarchitectural simulation and also do not cause any symptom in the functional simulation, we compare the application output with the fault-free output. The cases that have different outputs are categorized as silent data corruptions (SDCs). (If the functional simulation yields the same output, the fault is considered to be masked by the application as discussed earlier.)

For SPEC workloads, Figure 4.4(a) shows that only 0.3% of the injected faults result in SDCs for faults in all structures but the FPU. Server workloads also have a similar 0.3% SDC rate as shown in Figure 4.4(b). This is a rather low number given our simple fault detectors, and shows that our symptom-based detection techniques are effective for these structures.

For the FPU in the system running SPEC, 9.8% of the injected faults result in SDCs, largely because FPU

computations rarely affect memory addresses or program control (which are most responsible for detectable symptoms). Thus, our results show that the FPU requires alternate (potentially higher overhead) mechanisms to our simple symptom-based detectors. While the SDC rate shown here is quite low, it can be reduced further. One way is to employ software-assisted invariant detectors. Moreover, a recent study with my colleagues investigates the notion of application-aware SDCs and shows that the true SDC rate is often lower. We discuss this work in Section 8.2.

4.6.2 Software Components Corrupted

We next focus on understanding which software components (application or OS) are corrupted before a fault is detected (within the 10M instruction window of detailed simulation). This has clear implications for recovery. If only the application state is corrupted, it can likely be recovered through application-level checkpointing (for which there is a rich body of literature). However, OS state corruptions can potentially be difficult – software-driven OS checkpointing has been proposed only for a virtual machine approach so far [18]. On the other hand, hardware checkpointing methods are capable of recovering both the application and the OS state; full recovery, however, depends on the detection latency (to be discussed in Section 4.6.3).

For each structure, Figures 4.7(a) and (b) shows for SPEC workloads and server workloads, respectively, the percentage of fault injections that resulted in only application state corruption, OS (and possibly application) state corruption, and corruption of neither the application nor the OS. The height of each bar is the percentage of faults injected into the given structure that resulted in a detected symptom.

Our main result here is that over 65% of detected faults for SPEC workloads and over 84% of detections for server workloads corrupt OS state before detection. As we observe that server workloads generally have high OS activity (50+%), the OS is highly likely to activate the underlying fault and gets corrupted. On the other hand, while we observe that SPEC workloads have less than 1% OS activity during fault-free execution, a large fraction of the faults corrupt the OS. In these faulty cases, we observe that the OS is first invoked through a non-fatal trap after the application activates the underlying fault (e.g., a TLB miss in SPARC). As the injected fault is persistent, the OS execution subsequently activates the fault and the OS state is corrupted. Because our results show that a large number of faults corrupt the OS before they are detected, this motivates the exploration of techniques that are capable of recovering the OS and/or fault-tolerant strategies within the

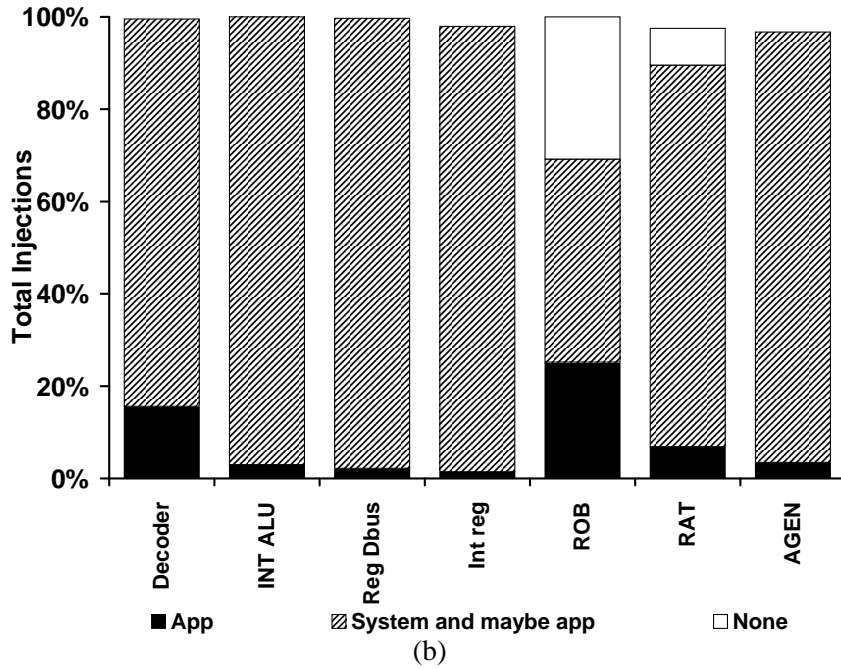
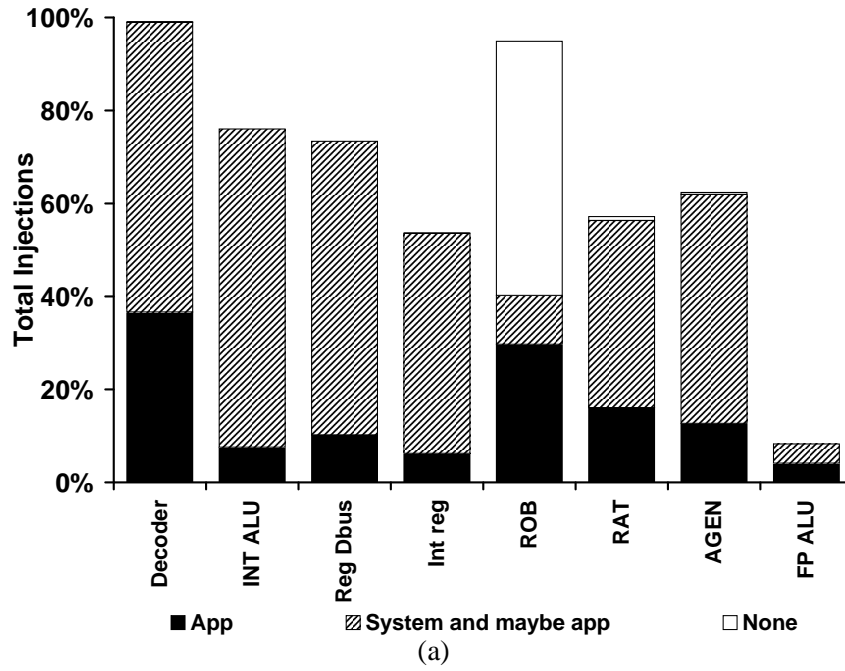


Figure 4.7: Application and system state integrity for the detected faults in (a) SPEC workloads and (b) server workloads. The height of each bar gives the percentage of injected faults detected in that structure. We see that most faults corrupt the system state.

OS.

We note that whether the application/OS state was corrupted is not necessarily correlated with whether the fault was detected at an application/OS instruction (discussed in Section 4.6.1). A fault could be detected at an OS instruction, but may have already corrupted the application state. Similarly, a fault could be detected in application code, but meanwhile the application may have invoked the OS resulting in a (so far undetected) corruption in the OS state.

Additionally, there are a few detected fault cases where neither the application nor the OS state is corrupted (58% of detected faults in the ROB and 2% in the RAT for SPEC workloads, and 31% of detections in the ROB and 8% in the RAT for server workloads). In all of these cases, the faults cause watchdog reset fatal traps to be thrown – the instruction at the head of the ROB never retires because its source physical register (say *preg_{head}*) never becomes available. These cases usually involve fairly complex interactions involving the ROB and the RAT. For example, consider a fault in the ROB that corrupts the destination field of a prior instruction that was supposed to write to *preg_{head}*. Because of the fault, the prior instruction writes to another physical register and never sets *preg_{head}* as available. If the corrupted destination was previously free, then this does not corrupt the architectural state (our implementation of register renaming records the corrupted destination name in the retirement RAT (RRAT) when the corrupted instruction retires, thereby preserving the architectural state).

4.6.3 Detection Latency

Detection latency is a crucial parameter since it affects recovery. Specifically, it affects the recovery strategy: the checkpointing interval, the amount of state that needs to be preserved for a checkpoint, and the cost of buffering for I/O. Small latencies allow the use of frequent but efficient hardware checkpoints and fast and complete recovery for both the application and the OS. Large detection latencies potentially require longer checkpoint intervals that result in longer restart on recovery and exacerbate the input and output commit problems. If the I/O commit problems are improperly handled, full recovery will be thwarted.

For each structure, Figures 4.8 and 4.9 report the histogram data on detection latencies (defined in Section 4.5.3) of all detected faults in SPEC and server workloads, respectively. The detections are categorized into the different stacks of the bars based on their latencies, ranging from 1,000 to more than 1 million

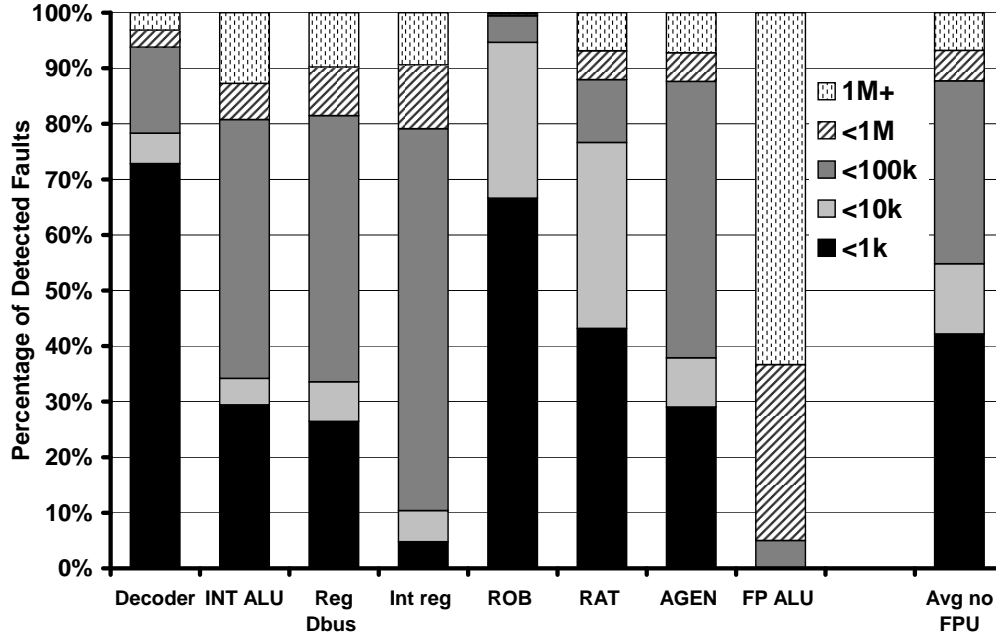


Figure 4.8: Total number of instructions retired from architectural state corruption to detection for SPEC workloads.

instructions. The height of each bar represents the total number of detections for the specific faulty structure.

While we are assuming for now that all faults detected within our 10M instruction window can be recovered with hardware checkpointing techniques, long detection latency puts more pressure on the I/O buffering mechanism. Interestingly, we find that most faults are detected much earlier than the detailed simulation window of 10 million instructions, with 87% of all detections occurring within 100,000 instructions for both SPEC and server workloads. This has implications for the type of checkpoints, the length of the checkpoint interval, and the I/O buffer size of the recovery mechanism. The impact of detection latency on the design of the recovery module will be explored in Chapter 6.

4.6.4 Transient Faults

From our transient fault injection experiments on SPEC workloads, as shown in Figure 4.10, over 96% of the injected transient faults were masked by the architecture or the application, with $< 0.4\%$ resulting in SDCs. Of the unmasked faults, 83% were detected by the hardware-only detectors.

For server workloads, Figure 4.11 shows that 90% of the faults were masked either at the architecture level or the application level. 59% of the unmasked faults were detected by the hardware-only symptom

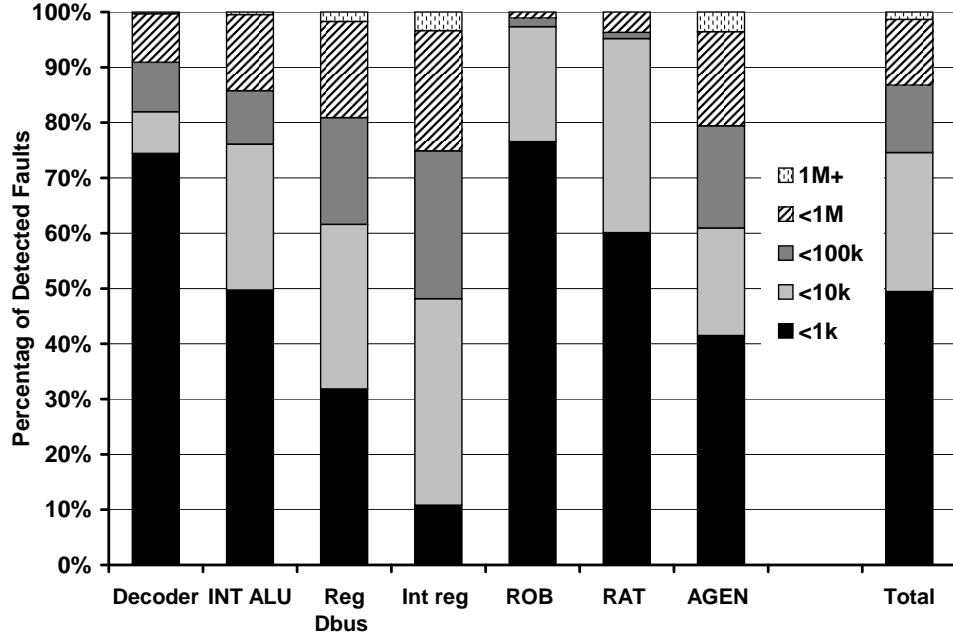


Figure 4.9: Total number of instructions retired from architectural state corruption to detection for server applications.

monitors. Faults injected into the RAT are particularly visible. This is due to the high utilization of architectural registers discussed earlier in Section 4.6.1. On the other hand, 4% of the injected faults result in SDCs. After a closer inspection, we found that this much higher rate of SDC is caused in part by the stringent setup of the workloads. In particular, the client applications are configured not to retry when an error is detected during the data transfer session with the server. As a result, the file in transmission is dropped and the resulting output differs from the fault-free output.

One may argue that the error is actually detected, albeit at the client system. In this thesis, however, we assume the sphere of recoverability (the logical extent of the system that is fully recovered by the underlying recovery method, discussed in Chapter 6) to be the server system only. Thus, if an error is not detected within the server system, we consider it a silent data corruption. Nevertheless, the next generation of SWAT can certainly take advantage of the inherent error-checking mechanisms at the network protocol level and the software level. For example, allowing retries after an error is detected in the network packet transmission can let the application mask the fault. We leave the exploration of protocol level error tolerance in SWAT for future work.

From these results, we found that both the very high masking rate and the coverage (i.e., percentage of

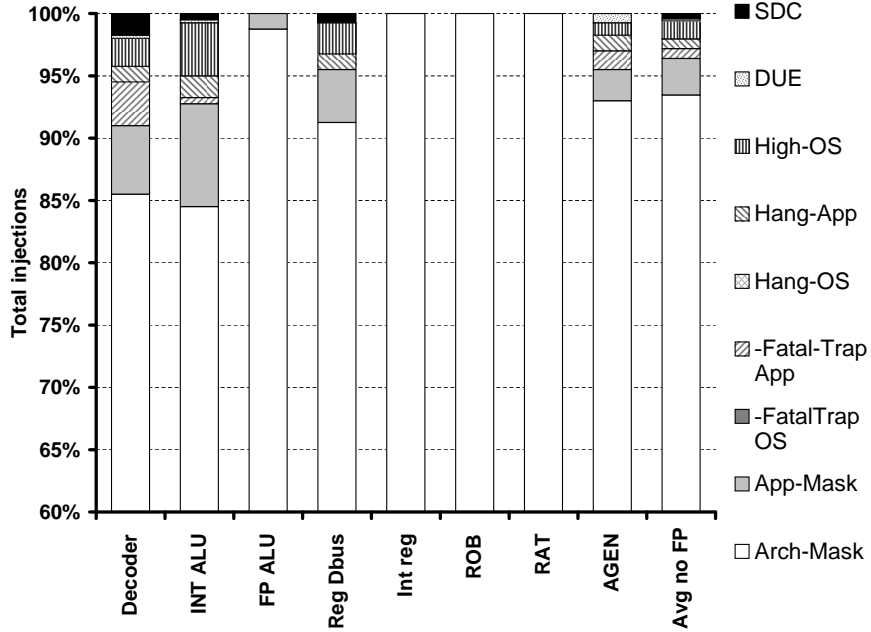


Figure 4.10: Coverage of SWAT hardware-only detectors for SPEC workloads on transient faults.

unmasked faults that are detected) of the symptom-based detection are consistent with previous findings [69, 81]. We note, however, that we rely on very simple and inexpensive symptom detectors to detect a large portion of these transient faults. More sophisticated low cost symptoms can also be employed to improve coverage. One such technique is our likely invariant detectors (results on permanent faults will be discussed in the following section). Further, the SDC rate of transient faults is a well-known problem. We show later that the use of invariant detectors can bring down SDCs significantly. One recent work of SWAT with my colleagues (not reported here) also looks into application awareness of SDCs [64]. We found that the true SDC rate is much lower than what is reported here when considering the error margins that are acceptable in many applications.

4.7 Results – Software-Assisted Detectors

We applied our invariant based detectors to five SPEC applications and permanent faults (a large body of application-specific detectors such as [54] has explored transients but very few look at permanent faults). Since likely invariants could result in detections that are false positive, we first present how the size of the input training sets affects the false positive rate. Then, the detection coverage, SDC rate, and overheads are

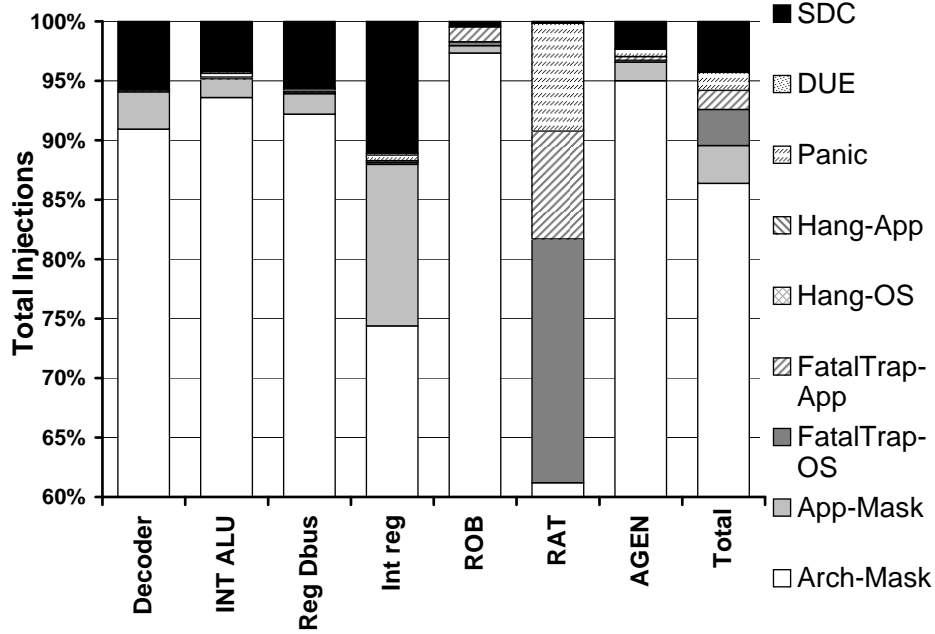


Figure 4.11: Coverage of SWAT hardware-only detectors for server workloads on transient faults.

compared between the workloads with invariant checking enabled and the same workloads with invariant checking disabled. To be consistent, the comparison is done on the same application binaries with invariant-checking code inserted. Hence, the binaries of these applications are different from the previously presented SPEC workloads. For the sake of convenience, we refer to the SWAT system with hardware-only detectors as hSWAT and the SWAT system equipped with invariant detectors as iSWAT.

4.7.1 False Positives

Figure 4.12 shows the variation of false positive rate (as defined in Section 4.5.3) for our five SPEC applications running on the ref input, as the number of training inputs is increased from 2 to 12.

As expected, false positive rate decreases as the number of inputs increases. By 12 inputs, the rate of false positives is less than 5% for all applications and 0% for three. This false positive rate is sufficiently low for our purpose, motivating us to use 12 training inputs for all of our experiments. In previous work using Siemens benchmarks [20, 54], hundreds of inputs were used for training. We find that much fewer training inputs suffice for permanent fault detection with our approach. These other proposed techniques try to keep the false positive rate as low as possible because these schemes cannot determine a false positive

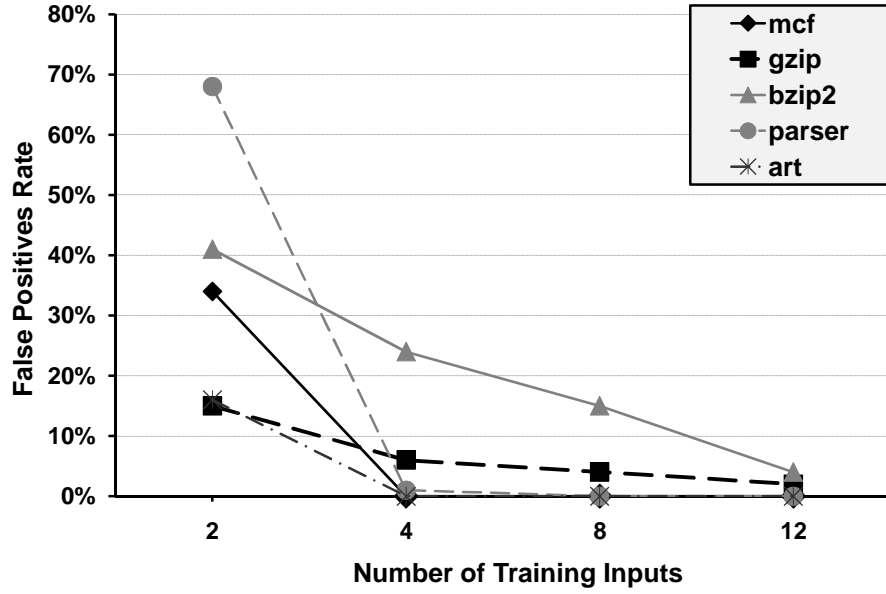


Figure 4.12: Variation of False positives rate with different number of training inputs. The rate is $<5\%$ with 12 training sets, motivating the use of 12 inputs for the rest of our experiments.

detection at runtime. If false positive detections repeatedly occur, the hardware is determined to be faulty and taken offline. On the other hand, because our invariant detectors can rely on the SWAT diagnosis module to identify false positives at runtime, our techniques is able tolerate more false positives.

The maximum number of static invariants in all applications was 231. Assuming each false positive detection has an overhead of 20 million instructions (considering that overheads due to rollback/replay of 10 million instructions and context migration), the maximum overhead of false positive detection on any input will only be 462 million instructions, which is negligible considering application executions that normally last for billions or trillions of instructions. In practice, the overhead will even be lower due to low false positive rates yielded from larger training input sets.

Interestingly, Figure 4.12 shows that after just four inputs, only less than 10% of the invariants are false positives for four applications. These results show that likely invariants generated from many inputs will have sufficiently few false positives, making it usable for permanent fault detection.

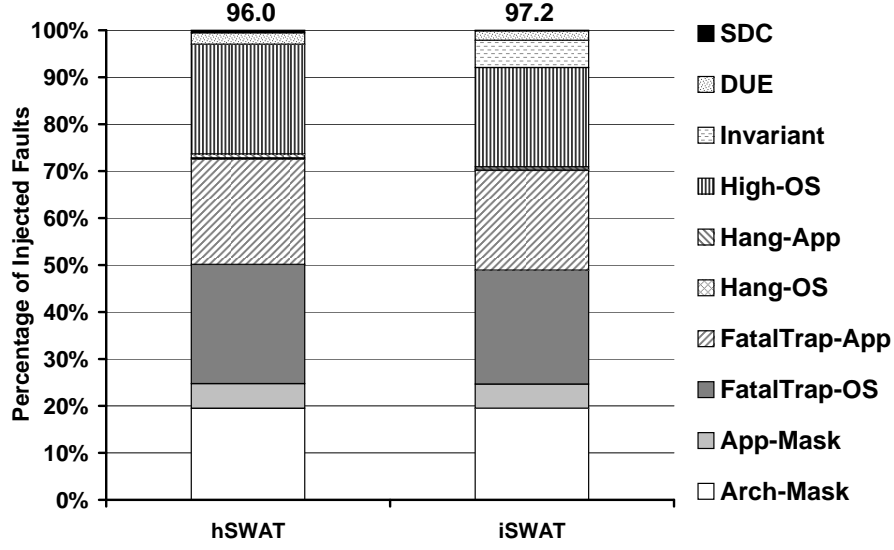


Figure 4.13: Permanent fault coverage of hSWAT and iSWAT.

4.7.2 Coverage

As discussed above, we use invariants derived from 12 training inputs for detecting injected permanent faults. Figure 4.13 presents the outcomes of the injected faults for hSWAT and iSWAT. The different stacks of each bar have the same categories as described in Section 4.6.1. The top of each bar shows the detection coverage achieved by the respective scheme.

From the figure, the overall coverage of the iSWAT system is 97.2%, improving from the 96% coverage of hSWAT. While the coverage increase seems small, there are three significant points that can be made from the results. First, the invariant detectors are catching nearly 5.8% of the total injected faults. Second, the invariant detection scheme is detecting some faults that are not detected by the hardware-only detectors, reducing the number of unrecoverable faults (DUEs and SDCs) by 28.7%. Third, the invariant detectors also detect some faults (about 5% of total fault injections) that are caught by the symptoms in hSWAT, but at a lower latency. This result leads to a small improvement in detection latency, as we show in Section 4.7.3.

Analysis of SDCs. Overall, the number of SDCs of the iSWAT system is significantly lower than that of hSWAT. The invariant detectors reduce the number of SDCs by **74%**, from **31** to **8**. We consider the reduction in the number of SDCs as the most important contribution of the iSWAT. Though a few SDCs remain, we believe that more sophisticated invariants can make the SDC cases negligible.

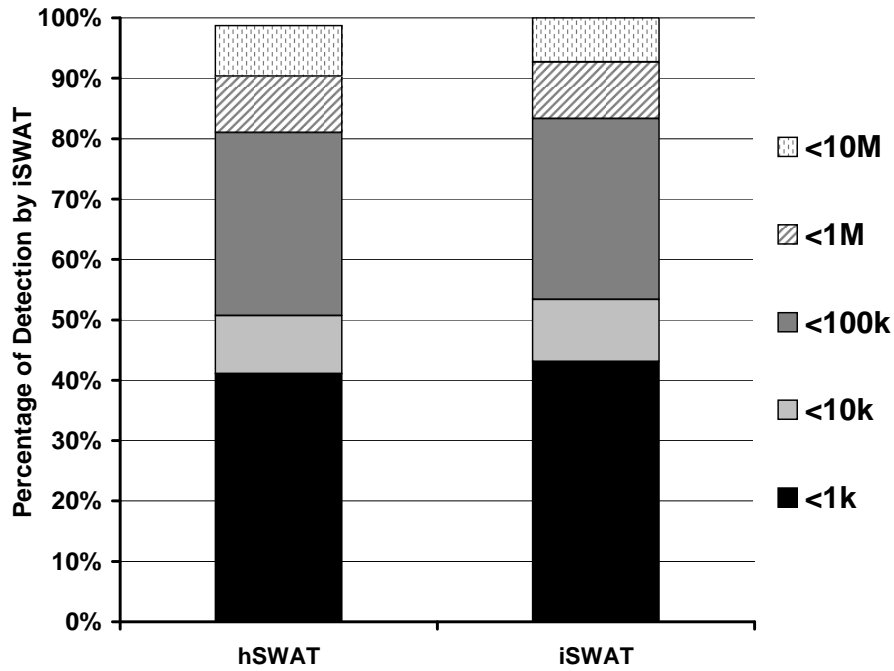


Figure 4.14: Detection latencies for hSWAT and iSWAT. The percentages are computed using number of recoverable detections in iSWAT as baseline. The invariants increase the number of faults detected within 1,000 instructions by 2%.

4.7.3 Latency

Figure 4.14 shows latency results for the faults detected by hSWAT and iSWAT, binned into various categories from under 1k instructions to under 10M instructions. In order to perform a fair comparison, the numbers are presented as a percentage of the total number of faults detected and recoverable by iSWAT (i.e. detections that happen within 10M instructions).

The number of faults detected at a latency of under 1k instructions shows the largest increase of about 2% (the rest of the numbers are cumulative). While the improvement seems incremental, this shows that invariants are able to slightly reduce the detection latency when compared to that of the hardware-only detectors. As these detections have very low latencies, they are amenable to simple hardware recovery mechanisms (Chapter 6 discusses the SWAT recovery strategy in greater detail). Although the latency benefits offered by iSWAT are not substantial so far, using more sophisticated invariants may improve the effectiveness of iSWAT to reduce the latency.

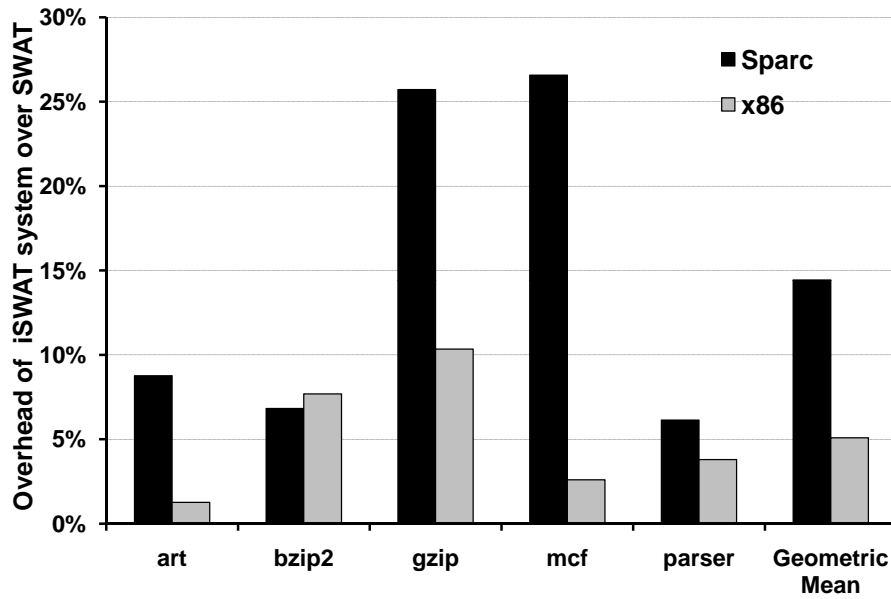


Figure 4.15: Overhead of invariants on an UltraSPARC-IIIi (Sparc) machine and an AMD Athlon machine (x86).

4.7.4 Overhead

We evaluate the overhead of using invariants by running the binary (with invariants checking) on fault-free hardware, using two machines: Sun UltraSPARC-IIIi 1.2GHz machine with 1MB unified L2, and 2GB RAM, and on an AMD Athlon(TM) dual-core MP 2100+ machine with 256KB L2 and 1.5GB RAM. The Sun machine is referred to as *Sparc* machine, and the AMD one as *x86* machine in this section.

Figure 4.15 shows the overhead of using invariants checking in the programs as a percentage over the baseline program which has no invariants checking. The geometric mean of the overheads is also shown for the two machines.

The Sparc machine exhibits a higher overhead when running the invariants code than the x86 machine, with the average overheads being 14% and 5% respectively. In particular, the overhead of the invariant checking mechanism in *mcf* is significantly higher in the Sparc machine (26%) than the x86 machine (2%). The high overhead of the Sparc machine is likely due to its inability to hide the cache misses and branch mispredictions induced by these extra invariant checks. The x86 machine that has a more sophisticated superscalar pipeline is able to hide these latencies better, resulting in lower overheads.

In spite of these differences, the overheads produced by these invariants checks are within acceptable overheads for the increased coverage that they provide, motivating the iSWAT system for increased error

resilience.

4.8 Summary and Discussion

This chapter introduces the key component of the low-cost SWAT resilient system – error detection. By employing monitors of anomalous software behavior, SWAT’s symptom detectors are able to effectively detect hardware faults that propagate into the software and appear as software bugs. This approach naturally handles all faults that matter and ignores the ones that do not. Since these symptom detectors can be implemented with near-zero hardware overhead, the cost of SWAT’s always-on error detection module is effectively minimized, optimizing the overall system cost.

Since SWAT takes a holistic approach to resilient system design, we can deploy more aggressive detection mechanisms in SWAT. In particular, we rely on the SWAT diagnosis to identify false positives and auto-tune the system if a false positive is diagnosed. Without involving the diagnosis mechanism, it will be much more difficult to use some of the symptoms described above. For example, it is hard to determine whether a detected hang is a false positive at runtime without replaying the execution on a fault-free core to see if the hang occurs again. Hence, by designing the system as a whole, the more aggressive detection mechanisms can be used and can help the system achieve high reliability.

Similar to other fault tolerant systems, the detection mechanism of SWAT also influences the design of the recovery mechanism. From our results, we find that permanent faults often corrupt the OS state, making OS recovery a high priority issue in SWAT recovery. By measuring the detection latencies of the injected faults, we found that a high percentage of faults can be detected within 100,000 instructions (approximately 100 μ s on a gigahertz processor), potentially allowing the use of simpler hardware recovery methods. Nevertheless, to fully recover all faults, sophisticated hardware rollback recovery mechanisms along with mechanisms that properly handle the input/output commit problems may be needed. We discuss the SWAT recovery strategy in detail in Chapter 6.

As the principle of SWAT detection is to watch for software anomalies, we can potentially leverage software reliability techniques for hardware faults. To this end, we took the first step in investigating the use of program invariants, a well-known software debugging method, to detect hardware faults. Our experimental results show that invariant detectors are very effective in detecting faults that mainly corrupt the data val-

ues of the program, successfully reducing the SDC rate of SWAT that uses only hardware-only detectors. Furthermore, as we rely on SWAT diagnosis to diagnose false positive invariant detections, more aggressive invariants (e.g., ones with higher false positive rates) can be used to ensure high detection coverage and low SDC rate.

Overall, through our experiments, we found that the SWAT detection approach is highly effective against hardware faults. While there is much work to be done to further improve the detection mechanism, the very low cost always-on detectors show great promise to ensure reliability for the mass computing market. A recent work with my colleagues (not reported in this thesis) already takes strides in this direction by investigating techniques for improving the detection coverage and latency, and deriving an application-aware metric for SDC that shows the SWAT system actually has lower SDC rates than ones reported here [64].

As mentioned earlier, the SWAT system is designed as a whole (instead of deriving various components independently). Hence, the diagnosis mechanism invoked post detection must be effective and be able to cater to the software-level symptom-based detection in SWAT. In the next chapter, we take an in-depth look at this SWAT diagnosis mechanism.

Chapter 5

SWAT Diagnosis

The device scaling induced hardware reliability problem has driven recent research to investigate the use of high-level detection techniques for deriving low-cost reliability solutions. Besides the software-level symptom-based detection of SWAT proposed in this thesis, some recent and contemporary work also investigates the applicability of high-level detection methods in hardware reliability (e.g., [17, 45, 54, 61, 81]). Such high-level detection mechanisms can be very effective because they provide coverage for a wide range of fault sources and faulty components.

While many of these proposals focus on transient faults where the detection and recovery components form the complete solution (e.g., a simple pipeline flush can recover from a transient error), emerging permanent faults require *diagnosis* in addition to detection. Because permanent faults are persistent, the faulty component must be diagnosed for repair/reconfiguration so that subsequent executions do not activate the underlying fault again and become corrupted. Only through a correct diagnosis, a system repair operation can be carried out by disabling the faulty component (such as a faulty core, ALU, or entries in a buffer, queue, or cache), reducing the frequency of operation of the component, or using software to replace the faulty execution of a specific instruction.

Although there has been significant recent work on high-level detection of in-field faults, there is relatively little work on diagnosing the source of a permanent fault detected in this way. The higher the level at which a fault is detected, the longer the latency between the actual fault activation and detection and the more difficult it is to diagnose its root cause for repair. Therefore, to reap the benefits of emerging low-cost high-level detection techniques, we need to develop effective diagnosis techniques. This chapter concerns such a diagnosis framework [35].

At a high level, the resulting SWAT diagnosis framework should fulfill two goals. First, because software bugs, hardware transient faults, and hardware permanent faults can all lead to software-level symptoms,

diagnosis must distinguish the type of fault the system is experiencing so that the correct action can be taken. For example, for a deterministic software bug, the diagnosis should allow it to propagate to the higher levels of software and become visible to the end user.

Second, in the case of a diagnosed permanent fault, the faulty component must be identified to the granularity of the field-reconfigurable unit to facilitate repair. The simplest repair solution would be disabling the faulty core [48]. That can be wasteful especially when modern superscalar processors often contain built-in redundancy (e.g., multiple decoders) that allows reconfiguring around failed components. Hence, the second goal of the diagnosis is to diagnose a permanent fault at the microarchitecture level to exploit this built-in redundancy in modern processors for repair, effectively recovering the system from permanent hardware faults.

Before deriving an effective method for achieving these stated goals, we first make the following key observations.

- It is acceptable to incur high overhead for the diagnosis procedure since the diagnosis is invoked only in the infrequent case after a fault is detected (in contrast, the detection mechanism needs to be low overhead since it must be on all the time).
- The faulty execution is known to cause an error detection. Hence, this same execution can effectively be used as a fault activating agent to assist the diagnosis process. (This is not unlike the modern functional tests for detecting faulty chips after manufacturing. However, this test is known to exercise the hardware fault.)
- The modern multicore environment provides a natural substrate for redundant execution. Diagnosis therefore can leverage this platform to intelligently trace the source of the fault.

Since the diagnosis can incur some performance overhead, we can use firmware to control the diagnosis process. The main advantage of this approach is that the firmware can conduct a more complex, intelligent analysis that would have been too expensive to implement in hardware. To precisely diagnose the cause of a detected error, the diagnosis firmware can observe the invariants when replaying the symptom-causing execution on different cores in a multicore system. In particular, our proposed diagnosis scheme has the following properties.

- Many detection schemes today rely on a checkpoint/restart mechanism for recovery [4, 45, 61, 81]. Our diagnosis relies on this mechanism to replay the execution that caused the symptom detection, effectively activating any persistent faults to give diagnosis clues.
- We exploit multicore systems by using a fault-free core to compare the execution with the symptom-causing core for the purpose of fault diagnosis. Effectively, we cheaply synthesize Dual-Modular Redundancy (DMR) for diagnosis, in contrast to expensive always-on DMR traditionally used for detection.

While SWAT diagnosis is proposed in the context of the SWAT system, this diagnosis framework, in reality, can work with different kinds of detection mechanisms and can be tuned to different granularity of repair. In the rest of this chapter, we first give an overview of the diagnosis scheme. Then, we discuss each of the two major diagnosis steps in greater detail. Since precise diagnosis is critical for fully recovering the system from permanent hardware faults (incorrect diagnosis would allow this type of faults to continue to corrupt the system), we show the effectiveness of the microarchitecture-level permanent fault diagnosis, TBFD, by presenting our experimental results. In the end, we summarize the lessons learned from the TBFD scheme and discuss the potential future work.

5.1 Diagnosis Overview

Our overall diagnosis scheme proceeds as follows. We assume a single-threaded program executing on a modern out-of-order superscalar core in a multicore system. We further assume a single core fault model, meaning that only one core is faulty in the system. The presence of the fault in the core is detected through the low-cost detection methods in the SWAT system as described in Chapter 4. (As mentioned, this detection can be triggered through other detection mechanisms [45, 61, 81].) After a detection, the SWAT firmware is invoked to perform the first step of the diagnosis, i.e., distinguishing among software bugs, transient hardware faults, and permanent hardware faults. By observing whether symptoms re-occur after repeated rollbacks/replays (provided by the recovery mechanism), the diagnosis is able to identify the source of the error. If a permanent fault is diagnosed, the diagnosis proceeds to the second step, i.e., identifying the faulty microarchitectural component. In our trace based microarchitecture-level diagnosis scheme, the SWAT

firmware rolls the faulty core back to a pristine checkpoint and replays the execution on it, while recording detailed information such as microarchitectural resource usage for all instructions. The SWAT firmware also transfers the checkpoint from the faulty core to a fault-free core and replays a “golden” execution on the fault-free core. The firmware then compares the traces from both the fault-free and faulty cores, and systematically analyzes the points of divergence to accurately diagnose the faulty microarchitectural structure.

In the following sections, we discuss each step of the diagnosis in more detail.

5.2 Diagnosing Software Bugs, Transient Hardware Faults, and Permanent Hardware Faults

While software bugs usually result in symptoms, transient hardware faults and permanent hardware faults can also manifest into the software and appear as symptoms. Because the handling of these faults is different, the first step of the diagnosis is to distinguish among them so that the correct action can be taken. For transient hardware faults (and non-deterministic software bugs), a simple rollback/replay to the last pristine checkpoint can fully recover the system from the error. For permanent hardware faults, before a rollback recovery, the diagnosis needs to identify the faulty component for repair/reconfiguration in order to prevent further fault activations, therefore system corruptions, in the future. For deterministic software bugs, SWAT lets them propagate to higher levels of software and become visible to the end user. Nevertheless, in future generations of SWAT, software reliability techniques such as Rx [60] can be used to handle deterministic software bugs, improving the overall system reliability.

While these faults may appear similar when they are first detected, the following observations help anchor the SWAT diagnosis strategy. Since transient hardware faults and non-deterministic software bugs only appear temporarily, re-executing from the previous pristine state will mask the errors. On the other hand, as both permanent faults and deterministic software bugs are persistent, simple re-executions will continue to lead to symptoms, making it difficult to distinguish between the two fault types. Nevertheless, if the persistence of a symptom is due to a permanent hardware fault, replaying the same execution on a different core in the system (as we assume a single core fault model, a core other than the faulty one is fault-free) would not result in a symptom. In contrast, a deterministic software bug will continue to cause a symptom

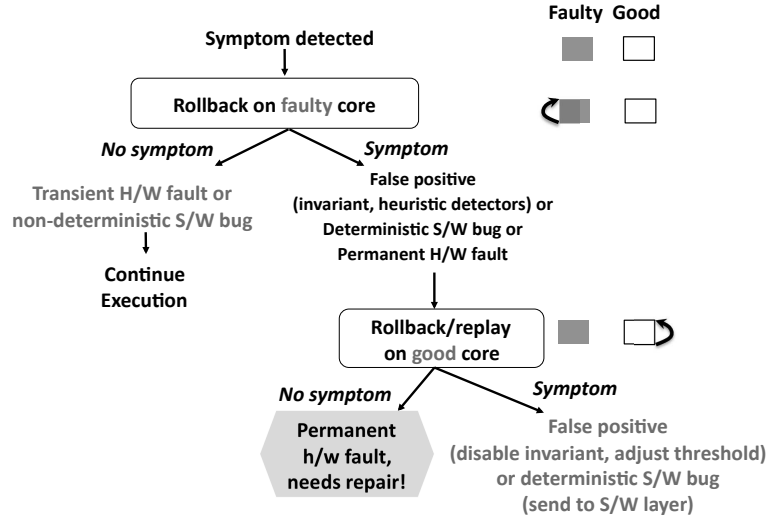


Figure 5.1: Diagnosis of a detected symptom. Through repeated replays, a software bug, a transient, or a permanent hardware fault is diagnosed.

in this new core.

Given the above observations, Figure 5.1 shows how SWAT diagnoses a software bug, a transient hardware fault, or a permanent hardware fault. The high-level idea of SWAT diagnosis is to watch for the re-occurrences of symptoms, if any, in repeated rollbacks/replays to determine the source of the error. During each replay, the diagnosis enters a phase we call the *vigilant phase* that determines if a symptom is a re-occurrence. A symptom is considered to re-occur if it is detected during the vigilant phase. In Chapter 4, we have found that most faults can be detected within 10 million instructions. We set the length of the vigilant phase to be three times of this maximum detection latency, i.e., 30 million instructions, which is often long enough for permanent faults to be activated and detected again. Nevertheless, this threshold can be configured to fit different system needs (e.g., future systems may have a much shorter maximum detection latency).

To distinguish different types of errors, after a symptom is detected, SWAT first rolls back to the previous pristine checkpoint and replays the execution on the same core. If a symptom does not occur again, the diagnosis concludes that a transient fault (a transient hardware fault or a non-deterministic software bug) is the cause of the previous symptom and resumes normal execution. We note that the rollback/replay naturally recovers the system from the transient error. On the other hand, if a symptom re-occurs (i.e., detected in the

vigilant phase), the diagnosis assumes this persistent symptom to be the result of either a permanent hardware fault, a deterministic software bug, or a false-positive detection of the heuristic detector (invariants, High-OS, hang, etc.).

Since a permanent hardware fault is the only error source among others that causes a symptom when the re-execution takes place on the same hardware, SWAT diagnosis transfers the checkpoint onto another (fault-free) core and replays the execution. If a symptom does not recur after this change in the hardware environment (executing on a different core), the diagnosis concludes that a permanent fault is present in the original symptom-causing core. The diagnosis algorithm reaches this conclusion because symptoms have occurred multiple times on the original core but disappear after the same execution is replayed on another core in the system. As discussed earlier, SWAT diagnosis also exploits the inherent microarchitecture-level redundancy to facilitate repair. Therefore, SWAT’s microarchitecture-level fault diagnosis algorithm, Trace Based Fault Diagnosis (TBFD), is invoked to identify the faulty component after a permanent fault is diagnosed in a core. We describe this method in Section 5.3.

On the other hand, a symptom can still re-occur after replaying on another core in the system. If the symptom is non-heuristic (e.g., fatal trap, kernel panic, etc.), SWAT diagnoses this as a deterministic software bug and lets this symptom propagate to higher levels of the software and become visible to the end user. If the symptom is detected by a heuristic hardware-only detector (e.g., High-OS, hang, etc.), the SWAT firmware considers the detection as a false positive and adjusts the threshold of the detector and resumes the normal execution. For example, if the High-OS symptom persists on both the original core and another core, the SWAT firmware suspects that the threshold value is too small. Consequently, the threshold is increased in order to prevent similar false-positive detections in the future. If the detection is an invariant violation, SWAT disables the static invariant to prevent future false-positive detections and resumes the execution.

We note that the SWAT firmware’s ability to identify false positives during runtime is essential to the overall SWAT system. If the SWAT system were not able to identify false positives online, we would have to resort to using detectors that are more conservative (e.g., sound range-based program invariants with larger ranges, higher High-OS threshold, etc.). Because of the presence of this diagnosis feature, more aggressive heuristic detectors can be used to potentially achieve higher detection coverage, hence higher reliability.

As multithreaded software is increasingly popular for taking advantage of multicore systems, we also

extended fault diagnosis to handle multicore systems running multithreaded workloads in other work (not reported in this thesis) [26]. We briefly discuss this work at the end of the thesis.

5.3 Diagnosing at the Microarchitecture Level

After a permanent fault is diagnosed in a core, one simple solution for repairing the system is to disable the entire core to avoid further corruptions by the fault. However, because modern processors already contain inherent redundancy, such as multiple functional units, registers, and so on, repairing at the finer grained microarchitecture level is possible. This level of repair is not only less wasteful than disabling the faulty core, but it also lengthens the lifetime of the faulty core. In order to facilitate microarchitecture-level repair, a diagnosis mechanism needs to be in place to identify the microarchitectural component that contains the permanent fault.

To this end, SWAT diagnoses at the microarchitecture level using a method we call *Trace Based Fault Diagnosis (TBFD)* [35]. TBFD is based on the following observations. First, the in-situ software execution can be used for activating the underlying permanent fault as the fault has already caused symptoms twice in the last diagnosis step. Second, the activated fault eventually leads to corruptions in the execution, which can be used as clues for diagnosis. Third, the multicore system provides a fault-free core that allows diagnosis to compare the faulty and fault-free execution for detecting corruptions.

The above observations drive the TBFD strategy: at the high level, the firmware-controlled TBFD exploits checkpoint/replay on the multicore architecture to inexpensively synthesize DMR for identifying divergences between the faulty and fault-free execution that provide clues for precisely locating the faulty microarchitectural component. Because the diagnosis is allowed to have higher overheads (since it is rarely invoked), we are able to use a firmware-based approach that provides the following benefits. By using firmware to conduct trace comparison, TBFD takes full advantage of the multicore environment without incurring the hardware overhead needed for lock-stepped execution in traditional DMR. Further, the firmware is capable of handling the more sophisticated trace analysis that is difficult to implement in hardware. One example is TBFD's capability for diagnosing meta-datapath faults (to be discussed in Section 5.3.2).

Now that we presented the TBFD approach, Figure 5.2 depicts how TBFD identifies the faulty microarchitectural unit X. The two main phases of TBFD are test trace generation and analysis of the test trace. In

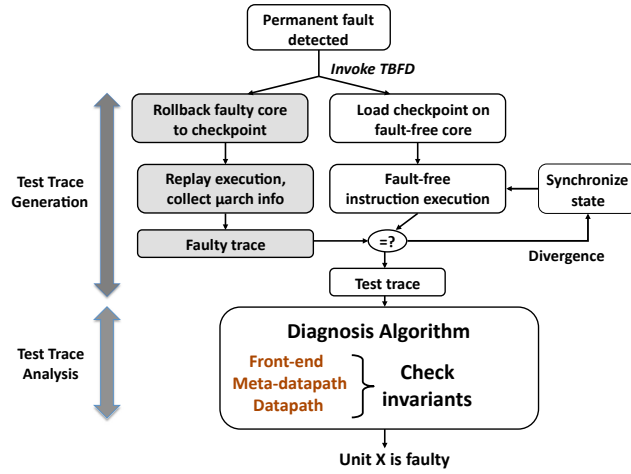


Figure 5.2: Diagnosis of a permanent hardware fault. By comparing the fault-free and faulty executions and analyzing the resulting test trace, the faulty microarchitectural unit is diagnosed.

the following sections, we discuss these phases in greater detail.

5.3.1 Test Trace Generation

As shown in Figure 5.2, TBFD compares the faulty and fault-free execution to generate the test trace. The test trace is essentially the execution trace of the faulty core (faulty trace) that is enhanced with microarchitectural resource usage information in the faulty core for each retired instruction and divergence information when compared to the fault-free execution trace (fault-free trace). In the following, we first describe how the faulty trace is obtained, then discuss how the test trace is generated.

Generating the Detailed Faulty Trace

To generate the *faulty trace*, TBFD rolls the faulty core back to the previous checkpoint and replays the execution for a predefined number of instructions. It records a trace of the execution with the following information for each retired instruction:

- *Decode*: Decoded opcode, immediate value, identifiers of source and destination logical registers.
- *Data values*: Values read from the source registers and values written into the destination registers. Virtual memory addresses accessed by loads and stores. Virtual target addresses of branch instructions.

- *Microarchitectural resources used by the instruction:* This category includes the source and destination physical register identifiers, the specific functional unit used, etc. The specific information recorded depends on the reconfigurable units supported in the processor and the consequent granularity of diagnosis required.

The decode information and the data value fields are used for comparing with the fault-free execution. If at least one of these fields of the retired instruction in the faulty trace is different from that of the fault-free execution, the diagnosis algorithm assumes that the permanent fault has been activated. Aside from identifying divergences from the fault-free execution, the hardware usage information recorded in the faulty trace is essential for the overall microarchitecture-level diagnosis. Specifically, during the test trace analysis (Section 5.3.2), the faulty instruction responsible for a divergence is tracked down (as we will discuss later, a faulty instruction may not result in a divergence immediately at retirement and the analysis needs to inspect a trace of instructions) and the algorithm deduces that one of the hardware resources used by this instruction must be faulty to cause the divergence. Thus, such hardware usage information is needed. Section 5.3.3 describes the hardware support for obtaining and recording the above information.

Fault-Free Execution and Test Trace

To obtain the *fault-free trace*, the fault-free core is loaded with the checkpoint of the faulty core and the execution is replayed. For each instruction in this execution, the TBFD firmware compares the decode and data value fields from the corresponding instruction in the faulty trace. Any mismatches in these fields cause the firmware to mark the corresponding instruction in the faulty trace as *mismatched* and record the field(s) that causes the mismatch. This event is important because it indicates that the execution on the faulty core has somehow activated the underlying permanent fault, which provides clues for tracking down the fault at the microarchitecture level. At this point, since the architectural state of the fault-free core is already different from the faulty core, retiring additional instructions in the fault-free core would lead to more divergences that are not caused by the activation of the permanent fault. To mitigate this, the firmware synchronizes (corrupts) the fault-free core's state to that of the faulty core. This allows the fault-free core to continue executing a path similar to the faulty core until the next activation of the fault.

Another possible scenario is when an instruction on the faulty core hangs at the head of the reorder

buffer and never retires because it waits for its source operand(s) indefinitely. The firmware marks such an instruction in the faulty trace as *hung*. We assume hooks are available to extract information of the hung instruction even though it does not retire. When a hung instruction is encountered, the analysis algorithm diagnoses the source of the fault by examining the test trace (Section 5.3.2). If the algorithm does not terminate after the analysis, both the faulty core and the fault-free core are rolled back to generate a new test trace for further analysis.

We refer to mismatched and hung instructions collectively as *misbehaved instructions*. We refer to the faulty trace enhanced with the information about misbehaved instructions as the *test trace*.

5.3.2 Test Trace Analysis

The heart of the TBFD algorithm is the analysis of the generated test trace to diagnose the fault. This analysis can be performed after completing building of the test trace. Alternatively, it may be periodically invoked after generating every N instructions of the test trace. The latter strategy may be more efficient if memory space to store the trace is at a premium. It also allows terminating test trace generation as soon as the diagnosis is able to uniquely identify the faulty structure.

TBFD divides the processor core into three different parts, on the basis of the information and analysis required to diagnose a fault in these parts:

1. *Front-End*: A fault in this part of the processor affects which instruction is executed, which operation is executed, and the logical source and destination registers accessed.
2. *Meta-Datapath*: Modern out-of-order processors use register renaming to translate logical register names to physical registers. Even if the front-end supplies the correct logical names, a fault in the translated name can result in erroneous computation. This type of fault is the largest source of complexity in TBFD – as we will show later, a corruption in the physical register name may not be caught by analyzing only the mismatched instructions. We use the term meta-datapath to refer to the parts of the core where a fault can corrupt the physical register name.
3. *Datapath*: This is the conventional data path, including the functional units, buses, and data residing in the physical register files.

In our work, we inject faults in the following structures as representatives of each of the above categories (see Table 4.3):

1. *Front-end*: Instruction decoders.
2. *Meta-datapath*: Register alias table (RAT) entries;¹ source and destination (physical) register identifier fields in the reorder buffer (ROB).²
3. *Datapath*: ALU, address generation unit, register data bus, and integer physical registers.

The TBFD test trace analysis described below assumes faults in only the above structures. Nevertheless, the algorithm can be extended to include other microarchitectural structures as well.

The analysis algorithm proceeds by using misbehaved instructions in the test trace as the starting point of the diagnosis. On encountering a misbehaved instruction in the trace, the algorithm systematically analyzes the misbehavior and determines if it can conclusively identify a fault in a unique structure. If so, it successfully terminates; otherwise, it updates counters corresponding to the microarchitectural resources used by the misbehaved instruction in the test trace. The algorithm then proceeds to analyze the next misbehaved instruction. If at any stage, one of the resource counters reaches a value higher than any other counters, the algorithm declares that resource as faulty and terminates. If the end of the trace is reached, the algorithm identifies the resources with the highest counter values as suspected faulty units – in this case, it is not able to uniquely identify a faulty resource.

Next we describe how TBFD systematically analyzes the misbehaved instructions to track down faults to the three targeted areas in the processor.

Faults in Front-End

If the misbehaved instruction is a mismatched instruction (i.e., not hung), TBFD first suspects a front-end fault. (As will be seen later, a hung instruction can only arise from a meta-datapath fault.) For this, it simply needs to check if the test trace indicates that the mismatch occurred in the decode information – such a mismatch indicates that the instruction word was corrupted at the front-end. For example, when the

¹We assume Intel Pentium 4 style register renaming with a distinct retirement register alias table or RRAT.

²In a real implementation, source register identifier fields would be in the issue queue; however, our simulator models them in the ROB and our algorithm uses the same terminology.

faulty instruction uses r_1 as source operand but the fault-free instruction uses r_3 as source operand, a fault is suspected in the front-end. Consequently, counters of the front-end units used in the faulty execution are incremented. In this study, since only decoders are accounted for in the front-end, the first mismatch in the instruction word makes the decoder used by the mismatching instruction identified as the unique faulty unit and successfully terminates the algorithm.

Faults in Meta-Datapath

If no front-end fault is found, TBFD analyzes both the mismatched and the hung instructions to check for meta-datapath faults.

This class of faults requires the most sophisticated analysis method. This is because, unlike the front-end and datapath, the first instruction that is affected by such a fault may not appear as a misbehaved instruction; i.e., it may not affect the fields in the faulty trace that are compared with the fault-free execution. Instead, it may silently corrupt the architectural state of the processor, causing later unrelated instructions to misbehave and obscuring the real source of the fault.

For example, in Figure 5.3, I_a writes to r_3 which is mapped to physical register p_{23} and I_c reads from r_3 . I_b writes to r_1 but is incorrectly mapped to p_{23} because of a meta-datapath fault (e.g., the register alias table had the wrong mapping). Thus, when I_b executes, r_3 is corrupted with the value of r_1 ; however, this is not indicated in any way in the information recorded for I_b in the test trace. Now when I_c retires, it sees the wrong value. This is caught when the faulty trace is compared with the fault-free execution and I_c is marked as a mismatched instruction. Now if TBFD were to blindly attribute this mismatch to the datapath structures used by I_c , the actual meta-datapath fault will never be identified.

In this work, TBFD focuses on meta-datapath faults in the ROB and RAT entries. In particular, TBFD checks the integrity of the logical-physical register mappings of the misbehaved instruction based on the following two conditions of fault-free executions.

1. *A non-free physical register can be mapped to at most one logical register at any time.*
2. *If an instruction reads from physical register p_x that is mapped to logical register r_y , the last instruction that writes to logical register r_y (the producer) must have written to physical register p_x .*

If a fault occurs in the meta-datapath, one or both of the above conditions may not hold. The first condition above handles the case discussed in Figure 5.3, where instruction I_c is detected as a mismatched instruction (step 1). To check if condition 2 is violated, TBFD searches backward in the test trace to verify the integrity of the mappings of I_c 's registers. The algorithm first looks for the instruction responsible for supplying the value of r_3 to I_c in the software. From this search, I_a is revealed as the producer of register r_3 that maps r_3 to physical register p_{23} (step 2). To verify that condition 1 holds, TBFD searches forward from I_a for the next writer to p_{23} but finds that I_b maps r_1 to p_{23} (step 3) while it is still mapped to r_3 (step 4). Thus, condition 1 is found to be violated. Since this event does not pinpoint where the fault is located, TBFD increments the counters of the RAT entries for both r_1 and r_3 and the ROB entry used by I_b . The RAT entry counters are incremented because a fault in the RAT entry can result in incorrectly mapping either r_1 or r_3 to P_{23} . Also, a fault in the destination register identifier field of the ROB entry used by I_b can map r_1 to p_{23} as well. While the source of the fault cannot immediately be known at this point, additional activations of the fault will cause the faulty structure to be involved in more violations of the above conditions. Consequently, the counter value of the faulty structure will be the highest among other suspected structures, allowing TBFD to precisely identify the fault.

Condition 2 is usually violated by a ROB fault. To check if condition 2 holds, TBFD goes backwards in the test trace from the misbehaved instruction to the producing instruction and verifies its logical to physical register mappings. For example, a fault in the destination register number field causes instruction I_A to write to a different physical register than indicated in the RAT. Then, a dependent instruction I_B reads the mapping from the RAT and waits indefinitely for a physical register that will never be set ready by I_A . As a result, I_B becomes a hung instruction. TBFD then starts tracing backward from I_B and finds I_A to be the producer. Because I_A writes to a different physical register than the one used by I_B , condition 2 is violated. As a result, TBFD increments the counter of the ROB entries of both I_A and I_B . The counters for both entries are incremented since a fault in either the destination physical register identifier field of I_A 's ROB entry or the source physical register identifier field of I_B 's ROB entry can cause this misbehavior. As mentioned earlier, with more misbehaved instructions, the faulty ROB entry can be uniquely identified.

However, even with techniques described above, RAT faults that are exercised by speculative instructions can be hard to diagnose down to the individual RAT entries. The scenario described below illustrates the

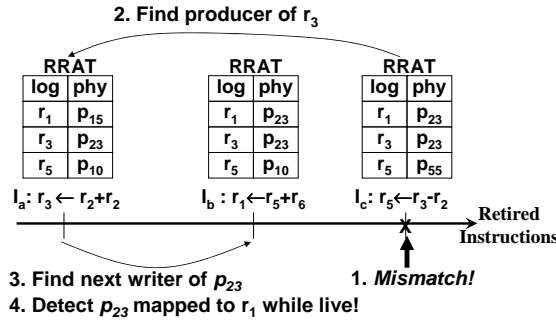


Figure 5.3: An example scenario depicting how a physical register that is mapped to more than one logical register is identified by TBFD.

difficulty. Consider that a logical register r_1 is mapped to a physical register p_1 . Suppose an instruction I that writes to logical register r_2 enters the rename stage. Because of a fault in the RAT entry, r_2 gets mapped to the already live physical register p_1 . Then, I executes, writes to p_1 , and wipes out the data in r_1 . Later on, I is squashed as a result of an exception or a branch misprediction, causing p_1 to be freed and added to the free list (even though it is supposed to be live and mapped to r_1). Subsequently, when another logical register is mapped to p_1 and written by another instruction that retires and becomes architecturally visible, r_1 now shows a corruption in the architectural state as its value is now incorrect. However, since TBFD never looks at the intervening speculative instruction I (remember that TBFD only tracks retiring instructions), the faulty RAT entry is not correctly identified. Nevertheless, as the execution continues to utilize the faulty RAT entry, more misbehaved instructions results. Subsequently, the test trace analysis shows that the faulty RAT entry is the direct or indirect cause of these misbehaviors, allowing TBFD to correctly identify the RAT faults.

Faults in Datapath

After TBFD determines that a mismatched instruction (the current TBFD fault analysis assumes that a hung instruction can only be caused by a meta-datapath fault) is unlikely to have been caused by a fault in the front-end or the meta-datapath, a fault in the datapath is suspected. At this point, the microarchitectural structures (the functional unit, the result bus, and the destination physical register) on the datapath that are used by the mismatched instruction are deemed potentially faulty. As a result, the counters of these structures are incremented. While a single mismatched instruction does not lead to a successful diagnosis (structures all have the same counter value), having multiple mismatched instructions can expose the faulty unit fairly

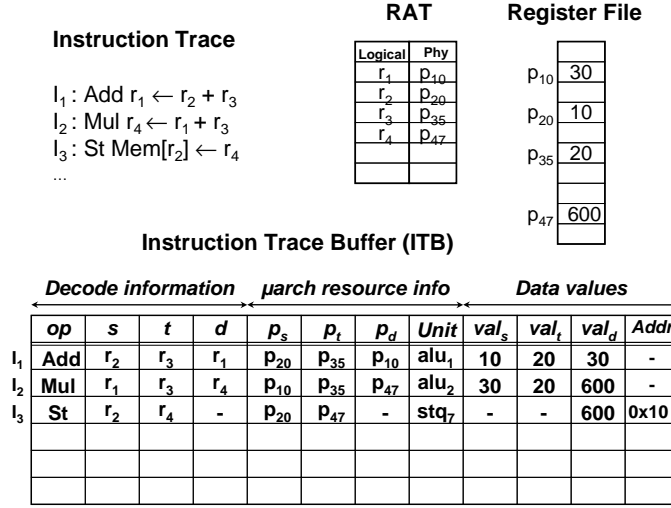


Figure 5.4: An example Instruction Trace Buffer (ITB). For each instruction retired by the faulty core during trace-based diagnosis, the ITB records information pertaining to 1) decoded instruction information, 2) some microarchitectural resources used by the instruction, and 3) the data values used by the instruction.

easily and result in an accurate diagnosis. This is because the faulty unit is involved in all these mismatched instructions and has the highest counter value.

5.3.3 Implementation

The TBFD algorithm is implemented in firmware. The detection of a fault on a core must result in an interrupt on another core (possibly through a protected channel) where the control transfers to the diagnosis firmware on that core. A single-core fault model implies that the latter core is fault-free; otherwise, the system must provide a protected, possibly simpler, fault-free core for the purpose of permanent hardware fault diagnosis and recovery. (Analogous support is likely required for multicore systems that aim to provide continuous operation in the presence of a non-repairable fault in a core.)

Additionally, the system must support checkpoint generation for the faulty core and checkpoint migration to a fault-free core. Several techniques have been proposed for checkpointing for the purpose of recovery from hardware failures [59, 74], and can be used for TBFD as well. For example, the ReVive scheme [59] could be used, with the checkpointed state made accessible to firmware on other cores.

The most significant hardware support required for TBFD pertains to the generation of the test trace. For this purpose, we propose to use an Instruction Trace Buffer or ITB, illustrated in Figure 5.4. Since

diagnosis is not performance-critical, the ITB could be implemented entirely in memory or in cache. For better efficiency, we propose an on-chip hardware FIFO buffer that is periodically flushed to memory.

On the faulty core, the ITB is responsible for storing three types of information for each retired instruction: the decoded instruction information, the microarchitectural resources used by the retiring instruction, and the data values of the retiring instruction. The decoded information of each instruction includes the instruction opcode, the source operands, and the destination operands. The microarchitectural resources usage information refers to microarchitectural structures (e.g., decoder, functional units, source and destination physical registers, etc.) that were used by the retiring instruction. The data values of the retiring instruction corresponds to the source register values, destination register value, the virtual address accessed by a load or a store, and the virtual target address of a branch. Figure 5.4 gives an example of an ITB for a small retirement trace from a faulty core. We discuss various issues related to the ITB and the test trace generation/analysis as follows.

Populating the fields of the ITB

Since the ITB is populated only in the rare event of a fault, we propose to populate the ITB with additional circuitry that taps into current microarchitectural structures for this information. An entry in the ITB is allocated once the instruction is decoded, with decode information from the decoder. When the instruction is allocated a ROB entry, and added to an issue queue, microarchitecture-level usage information (such as the physical registers used, ROB entry occupied, ALU used, etc.) can be populated. When the instruction writes its result, the data values corresponding to the instruction (destination register value and address) can be stored. If, however, the instruction is flushed, the corresponding entry from the ITB must be discarded as the trace accounts only for retiring instructions.

While the ITB and its upstream and downstream logic would incur area overhead, they are only activated during diagnosis after a rare event of a detection. During normal fault-free execution, these modules can be power-gated to minimize the power and performance overhead incurred by the diagnosis module. This is in contrast to previous methods of obtaining such information by adding wires that flow along with the instructions throughout the pipeline [11], consuming power and impacting performance during normal execution.

Diagnosis granularity and size of ITB

The granularity at which TBFD can diagnose a faulty microarchitectural unit is governed by the level of detail of the information recorded in the ITB. The fields to record in the ITB can be determined based on the level of repair supported by hardware. For example, if the hardware only supports replacing an entire array, as opposed to deconfiguring individual entries in the array, the ITB needs to only record the fact that this array was accessed, and not the specific entry in the array that was accessed. Not surprisingly, the finer the granularity of the repair or reconfiguration mechanism supported by the processor, the larger the ITB needs to be for storing these detailed information. In our experiments, we assume that fine-grained reconfiguration is supported for the parts of the front-end, meta-datapath, and datapath which may contain faults (Section 5.3.2), and TBFD records their usage information in the ITB.

Test trace generation and analysis

While the ITB can be configured to write the faulty trace into the cache or the memory directly, the fault-free execution trace has to be compared against the faulty trace during test trace generation. To this end, the fault-free execution is emulated by the firmware. Specifically, the TBFD firmware first loads the checkpoint from the faulty core onto the emulated core and then replays the emulated execution on the fault-free core. During the emulation, the firmware continuously compares the decode and data value fields against the faulty trace. On a misbehaved instruction, the firmware corrupts the architectural state of the emulated fault-free core and enhances the in-cache or in-memory faulty trace with bits to indicate the source of the misbehavior, thereby generating the test trace. These bits are best implemented as extensions to the ITB, indicating the sources of divergences in the test trace. Since the fault-free execution is already emulated in software, it is unlikely to benefit from any acceleration due to hardware FIFO support of the ITB. Therefore, the additional bits above need not be implemented in the hardware FIFO for the ITB, and can simply be maintained in cache or memory, depending on the particular implementation.

Finally, with the ITB containing the generated test trace, the trace analysis algorithm (part of the TBFD firmware) is invoked to diagnose the faulty microarchitectural component. This algorithm, implemented entirely in software, runs on the fault-free core in the system and conducts precise diagnosis by going through the test trace in the ITB.

5.3.4 Alternative Strategy for TBFD

The TBFD description above suggests that the fault-free core's state is synchronized to the faulty core's bad state when a mismatch occurs between the two executions. Essentially, during diagnosis, the execution on the fault-free core is *corrupted* with the state in the faulty core upon an occurrence of a mismatched instruction. Alternatively, we also considered another method where the faulty core is synchronized to the fault-free core's good state when a mismatch is encountered. We refer to this alternative as the *patching* (versus *corrupting*) execution.

One possible advantage of the patching execution over the corrupting execution is that the faulty execution is steered, through patching the faulty core with the good state, towards the correct path of the program. In contrast, the corrupting execution mode potentially steers both the fault-free and the faulty cores to random regions of code and data, which may or may not cause fault activations during diagnosis. We did, however, implement the patching method but did not find it achieving a better diagnosis coverage than the corrupting method. This finding implies that the corrupting method is capable of activating the underlying fault to enable precise diagnosis. Between these two execution modes, the corrupting method is favored because of the following reasons.

In the corrupting method, we do not have to execute the fault-free and faulty cores in synchrony. In fact, the entire faulty trace can be generated before the fault-free core starts execution. The firmware on the fault-free core takes care of corrupting the fault-free execution. In the patching method, this is not possible because the firmware cannot run on the faulty core. The faulty core must instead run roughly synchronized with the fault-free core. It must send the results of its instructions to the fault-free core and the fault-free core must send back any patches if needed. This is clearly much more complex and incurs higher overhead than the corrupting version. In addition, it requires the faulty core to patch the register file with data from the fault-free core while not knowing whether the path for overwriting the register file is fault-free.

It is interesting to note that the patching mode closely resembles the scheme proposed by Bower et al. where the DIVA checker is essentially the fault-free core that patches the architectural state of the faulty core [11]. While this is feasible in a tightly coupled scenario like DIVA, in a general multicore environment, it requires too *tight* lockstepping of two cores to be widely deployable.

5.4 Methodology

The goal of our experiments is to show the effectiveness of TBFD on permanent hardware faults. Correctly diagnosing permanent faults is important because a wrong diagnosis would allow a fault to corrupt subsequent execution, compromising the integrity of the system.

For our experiments, we use the simulation environment described in Section 4.4.1. We focus on SPEC workloads and apply TBFD to identify the components that contain the permanent hardware fault.

In this section, we describe the experimental setup that is specific to fault diagnosis.

5.4.1 Faults Diagnosed

TBFD is invoked to diagnose the injected permanent faults that are detected by the SWAT symptom monitors. In particular, approximately 8500 detected faults in the SPEC workloads (98% of the unmasked faults) out of the injected 11,200 stuck-at-0, stuck-at-1, dominant-0 bridging, and dominant-1 bridging faults (described in Section 4.4.2) in 7 of the 8 microarchitectural components (all except FP ALU in Table 4.3) are subject to diagnosis using our TBFD algorithm to identify the faulty microarchitectural component. We did not diagnose the FP ALU faults because there are only very few of them detected by the symptom monitors. More detections in the FP ALU is needed for the results to be significant. Nonetheless, our reported results are statistically significant as the overall error at a 95% confidence is a low 0.3%.

5.4.2 Implementation Assumptions

Our evaluation centers on the diagnosability of the TBFD approach. After all, a diagnosis method has not much value if it does not achieve necessarily high diagnosis coverage. To investigate this aspect, we enhance the base simulation platform to provide the microarchitecture-level diagnosis capability. We discuss these issues below.

Emulating fault-free execution: Since this is the first work that investigates the SWAT approach, we focus on single-threaded applications and do not simulate a multicore system. To obtain the execution of a fault-free core for TBFD, we exploit the inherent dual execution mode of the timing-first simulation paradigm in our simulation environment, as described in Section 4.4.1. When a fault is detected, the faulty core is rolled back to the previous checkpoint and the execution is replayed in the GEMS timing simulator; this roll-

back/replay would also happen in a real system. For the fault-free execution, we use the Simics functional simulator that runs in parallel with the timing simulation. Since the architectural state of the functional simulator and that of the timing simulators are compared for each retired instruction in the timing-first simulation paradigm, a mismatched instruction can be immediately detected by TBFD during retirement. Subsequently, the corrupted state in the timing simulator is copied to the functional simulator for synchronizing the fault-free and faulty execution. As the comparison and synchronization between the faulty and the fault-free execution can be done whenever an instruction retires, the test trace is generated as well.

Checkpointing: In our simulations, fault-free checkpoints are recorded at the beginning of the execution, prior to fault injection. To restore the previous checkpoint, our simulated system reloads the previously saved register state and TLB state, and rolls back (undoes) the changes in the memory state (similar to ReVive [59] and SafetyNet [74]). To ensure the execution on both the faulty and the fault-free cores to be the same when the fault is not activated, we also checkpoint the TLB state since the TLB in the SPARC architecture is software-managed. Otherwise, when re-executing from the same checkpoint, the out-of-sync TLB state could cause one core to drop into the OS to handle a TLB miss but not the other, leading to a false divergence.

Trace length: We run the faulty and the fault-free executions for up to 30 million instructions from the checkpoint. We empirically chose this interval since most permanent faults were re-activated for diagnosis. For efficiency, we invoke the TBFD analysis every 10,000 instructions. This buffer size is sufficiently large for the analysis to track down the faulty component of most injections.

Terminating conditions: After the analysis is invoked, if the algorithm is able find the unique faulty structure that has the highest counter value than any other units within 30 million instructions of execution, TBFD terminates and reports the identified faulty component. For some cases, if two units continue to have the same counter value until the end of the simulation, TBFD reports these two suspected faulty units. Nevertheless, the unique faulty unit can easily be diagnosed by deconfiguring one of the two units. On the other hand, due to the complex manifestation of the meta-datapath fault, there can be scenarios where three or more units with the highest counter value are RAT entries. For these cases, TBFD concludes that the RAT array is faulty (we discuss these cases in greater detail in Section 5.5.4).

Overall, the described simulation environment allows us to evaluate the TBFD approach quantitatively on the faults detected by the SWAT symptom monitors. In our experiments, we focus on the diagnosis

coverage of TBFD, which is defined as the percentage of detected faults that are diagnosable. Diagnosable faults are ones where TBFD is able identify the unique faulty unit (either a non-array unit such as an ALU, or an array entry), two potentially faulty units, and the faulty array structure (instead of the array entry).

5.5 Results

To understand whether TBFD is capable of diagnosing permanent hardware faults at the microarchitecture level, we investigate the diagnosis coverage of TBFD on the faults detected by the SWAT detectors. The results are shown and summarized in Section 5.5.1. From Section 5.5.2 to Section 5.5.5, we discuss in detail the faults that fall into the different diagnosis outcomes. While the diagnosis procedure has less timing constraint, it should still incur necessarily small performance overhead in order not to impact user experience. Hence, we report the diagnosis latency in Section 5.5.6.

5.5.1 Summary of Diagnosis Coverage

Figure 5.5 presents the results indicating the effectiveness of the diagnosis for faults in different microarchitectural structures. In each bar, the *Unique* stack represents cases that the diagnosis process correctly and uniquely diagnoses the faulty non-array structure or the faulty entry within an array structure. The *Among 2* stack represents cases that TBFD diagnoses 2 potentially faulty units and one of them is truly faulty. The *Correct Type* stack shows the cases where the diagnosis does not diagnose the faulty array entry (e.g., RAT entry), but the faulty array structure (e.g., RAT) is correctly diagnosed. The *Undiagnosed* stack represents cases where no misbehaved instruction is found for 30 million instructions. The *Incorrect* stack shows the cases where the diagnosis process diagnoses one or more structures as faulty, none of which is the actual faulty structure. The height of each bar is normalized to all the cases on which the diagnosis procedure is invoked (i.e., all faults detected within 10 million instructions as discussed in Section 5.4.1).

Of all detected faults, our trace-based diagnosis is able to diagnose 98% of the detected faults by identifying the correct faulty structure or array entry. Further, it correctly narrows 89% of the faults down to a single non-array structure (e.g., ALU) or a specific entry in an array structure (e.g., physical register # 15).

In the following sections, we give an in-depth analysis of the faults that fall into the different categories and discuss methods for improving TBFD even further.

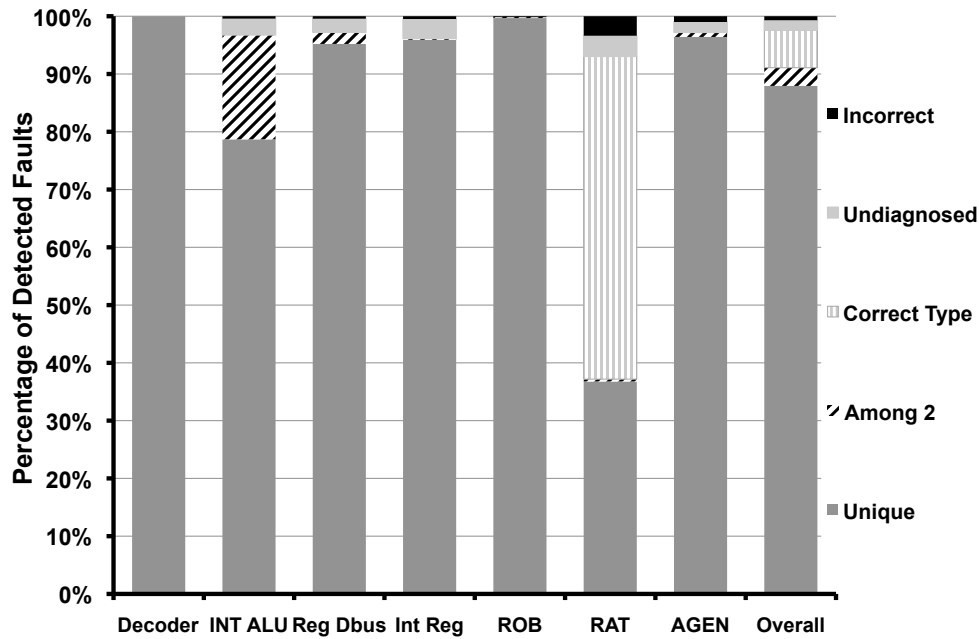


Figure 5.5: Effectiveness of microarchitecture-level fault diagnosis. The figure shows the ability of the diagnosis algorithm to accurately diagnose detected faults. Overall, 98% of the detected faults are accurately diagnosed as either (1) the correct non-array structure or the correct entry within an array structure (the Unique stack); or (2) within one of two non-array structures or entries of array structures (Among 2); or (3) the correct array structure type but not the correct entry within the structure (Correct Type).

5.5.2 Uniquely Diagnosed Faulty Structures

When TBFD correctly narrows a detected fault down to a single unit or array entry, we categorize the fault as uniquely diagnosed. While 89% of all detected faults can be uniquely diagnosed, from Figure 5.5, we see that different microarchitectural structures have varying amounts of uniquely diagnosed faults.

For 5 out of 7 structures (excluding INT ALU and RAT), over 97% and up to 100% of the detected faults are uniquely diagnosed; this shows TBFD is highly effective for diagnosing faults in these structures. In particular, virtually all the faults in Decoder can be uniquely diagnosed. This high percentage is likely due to the specific instruction word check in the first part of the diagnosis algorithm. Furthermore, over 99.6% of the ROB faults are uniquely diagnosed. From the diagnosis of ROB faults, we find the meta-datapath check is very important since most of these faults exercise this meta-datapath checking part of the TBFD algorithm.

For INT ALU, only 79% of the faults are uniquely diagnosed. The lower percentage is mainly due to the correlations with other structures (discussed in Section 5.5.3).

For RAT, however, only 45% of the faults can be uniquely diagnosed. While TBFD seems less effective for diagnosing faults in RAT, we note that without checking for faults in the meta-datapath, all of the RAT faults cannot be correctly diagnosed. Also, for array structures such as the register file, ROB, and RAT, there are existing testing techniques such as BIST (Built-In Self-Test) in the processor. Thus, TBFD may not need to diagnose the fault down to a single RAT entry, as long as it identifies the RAT as the source of the fault (discussed in Section 5.5.4).

5.5.3 Non-Uniquely Identified Faulty Structures

Since the diagnosis only analyzes the faulty core’s test trace and does not reconfigure the faulty core, if a correlation among two structures exists during execution, the diagnosis may not be able to uniquely diagnose the faulty component. The *Among 2* category reflects such cases where the diagnosis diagnoses 2 suspects that are potentially faulty, with one of the suspects being the structure with the fault.

Overall, only 3% of the diagnosed faults fall into the *Among 2* category. Most of them are faults in INT ALU (18% of INT ALU faults). A closer look at the *Among 2* cases shows that all mismatching instructions that use ALU 1 always write to their registers using Reg DBus 1. It is therefore virtually impossible to separate ALU 1 from Reg DBus 1 in our high-level trace-based diagnosis.

However, by narrowing the faults down to 2 non-array structures or array entries, the TBFD firmware can be enhanced to start another round of diagnosis after reconfiguring the microarchitecture. In particular, if the non-faulty components is disabled and TBFD is rerun, the faulty component will show up in this second round of diagnosis and be uniquely identified. On the other hand, if the faulty component is disabled, this second round of diagnosis will be non-conclusive and the disabled component is ruled faulty. Therefore, for *Among 2* cases, the faulty component can always be correctly and uniquely diagnosed in the second iteration of TBFD.

Alternatively, to reduce faults in the *Among 2* category, designers can break the correlations among resources by explicitly changing the scheduling algorithm in the processor. For example, Bower et al. implements a round-robin scheduling algorithm in the microarchitecture so that hardware resource usage is not always correlated [11]. Nevertheless, this approach likely has a non-negligible impact on area and power.

5.5.4 Faults Diagnosed in Higher Granularity

While TBFD is able to narrow down most of the faults correctly to one or two structures/array entries, only 45% of the detected RAT faults fall into *Unique* and *Among 2* categories. Such low percentage is mainly due to the reasons discussed in Section 5.3.2.

Although TBFD does not seem to perform well for RAT, we argue that it may not always be necessary for TBFD to diagnose to the exact RAT entry. As BIST based techniques that test array structures are increasingly common in modern processors (for manufacturing testing), it is useful to use TBFD to diagnose the RAT (instead of a particular RAT entry) as potentially faulty. Subsequently, BIST can be used to track down the actual faulty RAT entry. Alternatively, since RAT keeps track of the mappings of all architectural registers, a well-crafted functional test can also be used to exercise the different RAT entries to diagnose the fault.

For these cases, TBFD serves as a first-order test to quickly converge on the faulty RAT array. If we assume that it is sufficient to diagnose faults at the granularity of an array structure, TBFD can correctly diagnose an additional 44% of detected RAT faults.

5.5.5 Undiagnosed and Incorrectly Diagnosed Faults

In the previous sections, we see that TBFD diagnoses most of the faults detected in SWAT. As shown in Figure 5.5, the rest of these detected faults fall under two categories - *Undiagnosed* and *Incorrect*. In both these cases, the TBFD algorithm is unable to accurately attribute the location of the fault that was detected.

Of all detected faults, 2% fall in the *Undiagnosed* category, where the instruction traces of the faulty and the fault-free cores do not differ. In these cases, the permanent fault is either not activated or activated but masked by the architecture. Consequently, the architectural state of the faulty core is never corrupted. Because of the lack of divergence between the faulty and the fault-free execution, TBFD cannot carry out its analysis. These faults may be diagnosed by collecting a longer execution trace (currently a maximum of 30 million instructions are analyzed) or by using existing deterministic replay schemes [52, 83] to re-create the fault effect that lead to its detection. However, due to non-determinism at the microarchitecture level (e.g., conditions that result in different scheduling, register mapping, etc.), we note that the permanent fault is not guaranteed to be re-activated during diagnosis.

Regardless of the reasons for the lack of fault re-activation, since these faults would not cause any symptoms during the vigilant phase, they are diagnosed as transient faults in the first step of the SWAT diagnosis. Because these permanent faults appear as transient faults in the system, they can be recovered through rollback recovery. As they do not affect the software execution, the SWAT microarchitecture-level diagnosis algorithm correctly ignores them, avoiding excessive overhead.

On the other hand, only 0.9% of the detected faults are mis-diagnosed by TBFD to be a fault in fault-free structures. Further, from Figure 5.5, we see that most of these faults are in the RAT. We observe that these RAT faults cause data corruptions and mislead TBFD to diagnose the datapath components as faulty. This is mainly due to the problem caused by speculative instructions that activate the RAT fault (described in Section 5.3.2). While this can lead TBFD to disable the fault-free structure, the continued execution after the diagnosis would activate the persistent fault again, triggering another round of microarchitecture-level diagnosis. As a result, TBFD would get another chance to diagnose the permanent fault, increasing the likelihood of a correct diagnosis.

While further investigation to evaluate the best techniques to reduce, or eliminate, these misdiagnosed faults is necessary to make a fool-proof diagnosis algorithm, even with these limitations, TBFD presents impressive results for microarchitecture-level fault diagnosis at a very low cost.

5.5.6 Diagnosis Latency

Besides the percentage of diagnosable faults, another metric that measures the effectiveness of our diagnosis is the latency. If the latency is too long (e.g., billions or trillions of instructions), the processors' (both the faulty and fault-free cores) down time may make TBFD unattractive when compared to other simpler techniques, such as core decommissioning.

Our simulation infrastructure does not have enough detail yet to determine the latency in terms of the execution time of the entire diagnosis module. Instead, as a proxy, we report here the latency in terms of the number of instructions that the faulty core executes between the start of our diagnosis (i.e., after the core is rolled back to the previous checkpoint) to the point where the fault is identified. Figure 5.6 shows this latency. The figure includes all the faults in the *Unique*, *Among 2*, and *Correct Type* categories in Figure 5.5.

Of all the diagnosed faults, 56% take fewer than 1,000 instructions, 78% are diagnosed within 10,000

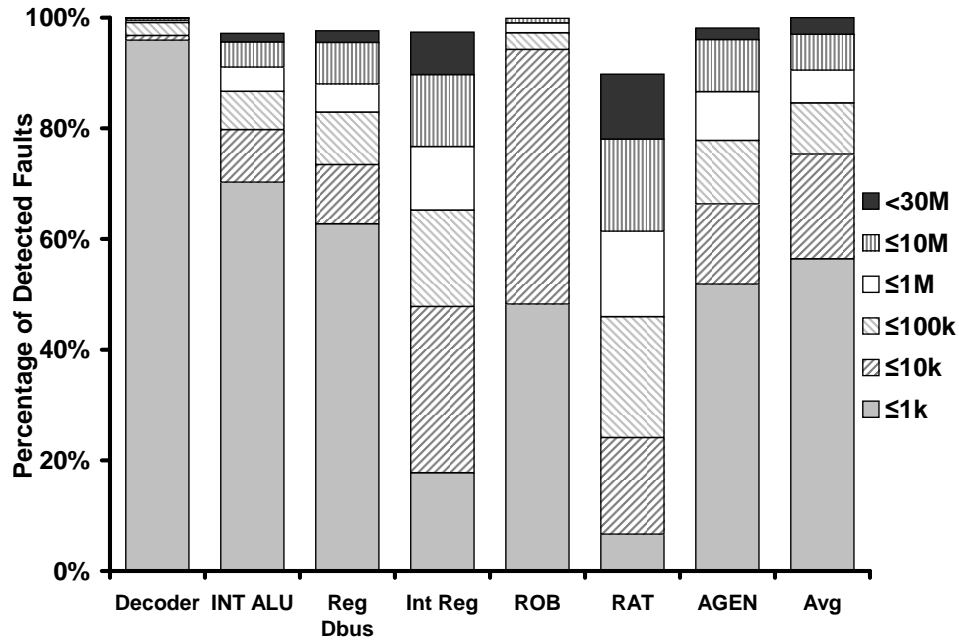


Figure 5.6: Diagnosis latency in number of instructions executed by the faulty core between the start of diagnosis and the point when the fault is diagnosed. The figure shows that over 90% of the faults can be diagnosed within 1 million instructions.

instructions, and over 90% take fewer than 1 million instructions to diagnose. Assuming a 4 GHz processor, these results show that close to 80% of the diagnoses occur within $2.5 \mu s$ and 90% of them terminate in $250 \mu s$. While we do not consider the overhead incurred by other aspects of TBFD, these overheads show that SWAT diagnosis can be short enough not to severely impact the system's response time.

From Figure 5.6, we see that the latency for faults in different structures varies widely. Over 99% of faults in Decoder and ROB take fewer than 1M to be diagnosed. The explicit check for front-end faults in TBFD helps shorten the diagnosis latency for Decoder faults. For ROB faults, the short latency is due to the fact that they usually cause a break in dependency and quickly lead to hardware hangs. This corresponds to the relatively quick violation in condition 2 discussed in Section 5.3.2.

On the other hand, only 77% of Int Reg faults and 61% of RAT faults are diagnosed within 1 million instructions. A general observation is that faults in these large array structures are more difficult to activate. The windowed register file in the SPARC architecture makes some RAT faults harder to activate because the faulty windowed register is rarely used. On the other hand, if the pressure on register allocation of an instruction stream is low, some faulty physical register may be infrequently mapped. These phenomena tend

to lead to longer diagnosis latencies.

As we stated earlier in the chapter, the SWAT diagnosis procedure is allowed to incur potentially higher overheads because it is infrequently invoked. From the results on latency shown here, we see that diagnosis does take some time to identify the faulty unit. Nevertheless, these diagnosis latencies are sufficiently small and are not likely to impact the overall system performance significantly.

Overall, in this section, our experimental results show that TBFD is very capable of intelligently tracking down the faulty microarchitectural units. This proves that permanent hardware faults can be precisely isolated at the microarchitecture level through test trace based analysis that makes use of in-situ software execution. Further, our results also show that the diagnosis latency is short and not likely to impact user experience.

5.6 Summary and Discussion

After a detection occurs in SWAT, the system must diagnose the source of the error because both hardware and software faults can appear as symptoms. Further, if a permanent hardware fault is diagnosed, full recovery can only be attained by repairing or reconfiguring around the faulty component to prevent the same fault from corrupting subsequent executions. This chapter presents the SWAT diagnosis module to resolve these issues.

Since diagnosis is only invoked after an infrequent event of a detection, such process can potentially incur higher overhead (in contrast to always-on detection mechanisms that must incur as little cost as possible). The SWAT diagnosis framework therefore takes a firmware-based approach that enables intelligent, possibly complex diagnosis procedures. As the in-situ software execution is known for activating the possible persistent fault (if any), SWAT diagnosis exploits the multicore environment by replaying this execution on different cores to track down the source of the error. Fundamentally, SWAT takes a synthesized DMR approach for the purpose of fault diagnosis, which incurs much lower cost than the traditional DMR that needs to be always-on for error detection.

In SWAT diagnosis, the first step of the process uses repeated rollbacks/replays on different cores in the system and watches for symptom re-occurrence, if any, to distinguish among software bugs, transient faults, and permanent hardware faults. If a permanent hardware fault is diagnosed in a core, the second

step of the diagnosis, called Trace Based Fault Diagnosis (TBFD), looks for divergences in the replayed executions on the faulty core and a fault-free core, and identifies the faulty microarchitectural component through execution trace analysis. This process essentially synthesizes the faulty core and a fault-free core in the multicore system for dual modular redundant execution to enable microarchitecture-level diagnosis. The main novelty of TBFD lies in its ability of identifying permanent faults that occur in the meta-datapath. To the best of our knowledge, no existing functional testing technique can properly handle this type of faults.

To evaluate our proposed microarchitecture-level diagnosis scheme, we used TBFD to diagnose the permanent faults previously detected by SWAT's symptom detectors and found that TBFD is highly effective, correctly diagnosing 98% of the detected faults down to the microarchitecture level. In particular, the unique faulty non-array structure or array entry is identified in 89% of the detections. The rest of the diagnosed cases belong to the following categories: (1) TBFD identified two units and one is faulty, and (2) TBFD is unable to identify the individual array entry but diagnoses the faulty array structure. For these cases, the unique faulty unit can eventually be identified by another iteration of TBFD after disabling one of the suspected units (for case 1) or using functional tests that target the specific faulty array (for case 2). In terms of performance overhead, TBFD is able to identify 90% of diagnosed faults within 1 million instructions, which approximately equals to 250us on a 4 GHz processor. We believe this latency is short enough not to substantially impact the overall system performance, especially when the diagnosis is rarely invoked.

In addition, TBFD is the first work that uses in-situ execution to diagnose faults in the meta-datapath. From our results, TBFD is able to diagnose a large portion of these faults down to the unique array entry and most faults at a higher granularity (array structure vs. array entry). These results are encouraging as our method remains effective even though the manifestation of meta-datapath faults is complex. Further, we believe that there is room for improving the TBFD algorithm to achieve even better diagnosis coverage for faults of this nature.

Overall, this chapter introduces the SWAT diagnosis framework to carry out effective fault diagnosis on the SWAT system. However, we note that this proposed framework is flexible and can easily be integrated with different types of error detection mechanisms. Further, the framework is also tunable to cater to different granularity of repair supported by the system. Since this is the first work to investigate the capability of the SWAT system, we have mainly focused on single-threaded workloads. As multithreaded software

and multicore systems are becoming pervasive, recent work with my colleagues (not reported here) investigated the diagnosis mechanism for multithreaded applications running on multicore systems [26]. In that work, we successfully derived an effective diagnosis strategy for attaining high diagnosability even in this multithreaded environment.

Chapter 6

SWAT Recovery

The goal of error recovery is to mask any effect of the detected error by restoring the system to a pristine state and resuming the execution. Hence, error recovery is vital to uphold the integrity and reliability of any fault-tolerant system, including SWAT. While there has been a wide selection of error recovery proposals that can be potentially integrated with SWAT, the effective method must be chosen carefully in order not to compromise the strength of the SWAT system.

Towards proposing a recovery scheme for SWAT, we must consider various issues. First, the recovery module should be designed to fit the needs of the other components in the system, namely the detection and diagnosis modules. For example, since the SWAT symptom monitors have certain detection latencies, our recovery must be capable of handling the effects of error propagation in the time period prior to a detection. Second, as all recovery methods incur overheads in area, power, and/or performance during fault-free operations, these overheads must be kept low in order not to impact the overall system cost significantly. Desirably, the recovery scheme should have a similar cost as the always-on very low overhead hardware-only detectors of SWAT.

In this chapter, we first give an overview of the design constraints of the SWAT recovery module. Then, we discuss checkpoint/replay mechanisms and I/O buffering schemes that SWAT recovery can leverage. After that, we explore the potential designs of the recovery module quantitatively in terms of system recoverability for faults previously detected by SWAT's symptom monitors. Based on this analysis, we investigate the overheads involved for designing the recovery scheme that can achieve high recoverability. While we do not propose a new recovery method in this thesis, the quantitative and qualitative analysis presented here can effectively guide the exploration of new lower-cost and more effective recovery mechanisms for SWAT and other error resilient systems.

6.1 Constraints and Requirements of SWAT Recovery Module

The SWAT system is able to achieve very low cost because of the key observation of minimizing the overhead of the *common* fault-free operation. However, as shown in Figure 3.2, error recovery incurs overheads in both fault-free operation and fault-handling operation. Therefore, to be consistent with the SWAT design principle, the fault-free operation of error recovery must be kept low cost.

Another important role held by SWAT recovery other than masking the error effect is to assist the diagnosis process. In SWAT diagnosis, the fault-activating execution is repeatedly replayed in order to correctly determine the source of the error. Therefore, SWAT’s recovery mechanism must be able to allow the system to “go back in time” to replay the execution.

Besides the above requirements, we also need to consider how hardware faults can be contained to ensure full system recovery. To this end, Reinhardt and Mukherjee introduce the *sphere of replication*, defined as the logical extent of redundant execution (in time or space) for protecting the system from soft errors [66]. At the boundary of this sphere, outputs are compared before they become visible to the rest of the system. Similarly, Sorin uses the *sphere of recoverability* to describe the logical extent of the system that is protected by SafetyNet [74]. To ensure full recovery, output events that cross the sphere of recoverability are only made visible to the rest of the system after the checkpoint is validated.

Like other systems, in SWAT, the sphere of recoverability depends on the employed recovery technique, which in turn is dependent on the error detection latency. If the detection latency is very low (e.g., under 100 instructions), processor-level checkpointing along with memory transaction buffering (using the store buffer) can be used to recover from an error. In this case, the boundary of the sphere of recoverability lies between the processor and the first-level cache. If the detection latency is very long, software-level techniques such as the ones used in distributed systems may be needed. The sphere of recoverability, hence, may cover multiple systems that are involved in running the distributed application.

In this thesis, based on the detection latency found in Chapter 4, we focus on the sphere of recoverability that contains both the processor and the main memory. Hence, the input/output (I/O) events that occur during the software execution must be properly handled until the system is validated to be fault-free. Otherwise, a faulty output request issued by the processor (e.g., a request sent to the network interface card) may become visible to the rest of the system before an error is detected. This scenario is problematic as the visibility of

the erroneous event is irreversible and full system recoverability cannot be achieved. Therefore, in general, output events are buffered and are not made visible to the devices until the state of the processor and memory is validated to be fault-free. For input events, some of the requests (e.g., disk accesses) can be regenerated by the processor without buffering while others would need a mechanism to buffer and replay the input events (e.g., a keystroke from the user) during recovery. In this chapter, we mainly focus on buffering output events as faults can effectively be contained this way.

With these requirements, the design of the SWAT recovery scheme can be subdivided into (1) providing a mechanism for replaying executions that incurs low cost during fault-free operation and (2) employing a scheme for buffering output events to contain faults. Both of these issues need to be addressed to ensure full system recovery.

In the following sections, we first discuss the mechanisms for execution replay and then investigate issues in I/O buffering.

6.2 Mechanism for Execution Replay

Error recovery can be broadly classified as forward error recovery (FER) and backward error recovery (BER). (Section 2.3 discusses related work in both FER and BER.) FER recovers an error by moving forward in the execution because the fault-free state is readily available in the system [46]. BER recovers an error by rewinding the system state backward to a fault-free state for re-execution. In terms of cost, FER tends to be higher than BER as it needs mechanisms for keeping a known fault-free state; in comparison, this makes BER more favorable. Also, by definition, BER provides support for replaying the execution, making it a suitable candidate for SWAT recovery since this is required for diagnosis. Hence, in the rest of the chapter, we focus on potential BER techniques that the SWAT recovery module can leverage.

6.3 Checkpoint and Replay Mechanisms

Checkpoint-and-replay is a well-known backward error recovery method. In these schemes, a checkpoint is established periodically so that the state of the system can be restored after an error detection. While checkpoint creation suggests that snapshots of the system state are taken, modern checkpointing schemes often use

	Hardware Checkpointing	Software Checkpointing
Hardware Overhead	Varies	None
Performance Overhead	Low	High
Checkpointing Frequency	High	Low
Recovery Overhead	Low	High

Table 6.1: Comparison of hardware and software checkpointing schemes.

a combination of checkpoint-based and log-based mechanisms for checkpoint creation. For example, the full register state of a processor is often saved as a snapshot due to its relatively small size while logging is used to store changes to the memory state of the system. In this thesis, we regard these methods collectively as checkpointing mechanisms. After a checkpoint is established, the system can be rolled back to the previous pristine state when an error is detected. The execution is then resumed and the effect of the detected error is masked.

Of all the previously proposed checkpointing methods, we broadly categorize these methods as software-based and hardware-based. Based on the requirements above, the different tradeoffs of the two types of checkpointing schemes are discussed below. Table 6.1 gives a brief comparison between these methods.

6.3.1 Software Checkpointing

Software checkpointing works by periodically creating checkpoints of the system state so that the system can be rolled back when an error is detected. Software checkpointing can also be categorized into application-level and system-level. While future SWAT systems may improve recoverability through application-level checkpointing, we focus our discussion on system-level checkpointing since it is more general and can be applied to different applications.

System-level software checkpointing typically relies on checkpointing of processor state and logging of memory pages through copy-on-write mechanism to establish checkpoints [33, 58, 76]. At the beginning of the checkpoint interval, all memory pages are marked read-only. Whenever a page is written to, a memory protection exception is triggered and the checkpointing mechanism makes a copy of the page before it is written to. Depending on the individual scheme, this copy of the memory page can be stored on a stable storage such as disk [33] or a volatile medium such as DRAM [58]. The collection of stored memory pages then forms the undo log of the memory and hence can be used to roll back to the previous checkpoint in the

presence of a fault.

The main advantage of software checkpointing is that it does not require hardware support because the checkpointing and rollback can be carried out entirely in software. However, the performance overhead for software to take a checkpoint (copying memory pages) could be quite large. Because of the high checkpointing latency, the checkpointing frequency is kept relatively low (e.g., once per 30 minutes) to amortize the overhead. Consequently, the large checkpoint intervals could complicate the handling of the input/output commit problems. Furthermore, in system-level software checkpointing, as the OS is responsible for creating checkpoints, additional mechanisms for recovering the OS are needed as hardware faults can corrupt the OS state. Towards this end, using software-based methods for recovering the OS is non-trivial and has only been done using virtual machine monitors (VMMs) [18].

Overall, the high overhead in performance during fault-free operation, the obstacles involved in properly handling I/O for very long checkpoint intervals, and the complexity in recovering the system software make software checkpointing unsuitable as the primary recovery mechanism for SWAT. However, in future generations of SWAT, software checkpointing can be leveraged to improve SWAT's recoverability for some faults that may have very high detection latencies.

6.3.2 Hardware Checkpointing

Similar to software checkpointing, hardware checkpointing uses a combination of checkpointing and logging techniques to establish checkpoints where the faulty machine can roll back to. Hardware checkpointing can be further classified as processor-level and processor-and-memory-level. Processor-level checkpointing refers to checkpoints that are created within the processor and are transparent to software. Pipeline flush, the built-in rollback mechanism in modern processors for handling branch mispredictions and exceptions, is a prime example of processor-level rollback recovery to the previous checkpoint established. As most, if not all, processors contain this mechanism, much prior work relies on pipeline flushes to recover the processor state from soft error detection [4, 61, 66, 68, 81]. As mentioned earlier, if the detection latency in SWAT is sufficiently short, processor-level rollback recovery along with memory transaction (loads and stores) buffering can be employed. However, from the results presented in Chapter 4, the maximum detection latency of 10 million instructions likely requires a non-trivial buffering mechanism for memory accesses.

Hence, we investigate processor-and-memory-level checkpointing methods.

Processor-and-memory-level checkpointing, as the name suggests, establishes checkpoints for the state of the processor and the memory. In these schemes, snapshots of the processor state are periodically taken and stored in an on-chip buffer (e.g., shadowed register file). On the other hand, checkpointing the entire memory state is very difficult given the size of today's system memory. Hence, many of the proposals resort to use logging for rolling back to the previous checkpoint. The logging technique used is similar to software checkpointing; a change in the memory state triggers the old state to be saved as an entry of the undo log. However, with hardware support, the logging of the memory state is typically done at the memory block (or cache line) granularity instead of the memory page granularity because the cache coherence protocol usually operates at the block granularity and can be enhanced for this purpose. To recover to a previous checkpoint, the logged memory blocks are used to patch the memory back to the state when the checkpoint was taken. Typically, the performance overhead incurred during the common fault-free execution is caused by the synchronization operations needed for establishing consistent checkpoints and the increased demand on memory bandwidth for creating the undo logs. Nevertheless, with dedicated hardware support, checkpoints can usually be taken very efficiently. Hence, the fault-free performance overhead can be kept low. This is particularly attractive since SWAT demands very low performance overhead for common case operation. Nevertheless, we must investigate the actual performance overhead and the cost of the added hardware support of the recovery scheme to determine if the method is sound for the SWAT system. If the common-case overhead in area, performance, and/or power is significant, the scheme would greatly increase the overall cost of SWAT.

Of the different processor-and-memory-level checkpointing methods, we look at two recently proposed techniques for recovering the state of multiprocessor systems: SafetyNet [74] and ReVive [59]. These schemes take periodic checkpoints of the processor and memory state of distributed shared memory systems. We discuss the use of these techniques for recovering the processor and memory state in the SWAT multicore systems.

SafetyNet

SafetyNet, proposed by Sorin et al., uses versioned caches and on-chip buffers called Checkpoint Log Buffers (CLBs) to efficiently checkpoint a distributed shared memory system [74]. By enhancing the coherence protocol and using time-stamps on each cache line or the corresponding memory block, SafetyNet is capable of establishing globally consistent checkpoints without interrupting the system’s operation.

Specifically, each cache line is marked with a time-stamp to identify the checkpoint it belongs to; a service processor periodically sends the current checkpoint number to each processor node. With the time-stamp, the processor can determine if a store issued to a particular cache line is the first modification since the checkpoint. If so, the pre-modified version of the cache line is logged in the CLB. Collectively, the CLBs form the undo log of the memory state and hence rollback recovery to the previous checkpoint is possible. From their experiments, with a total CLB size of 512KB, SafetyNet can take a checkpoint every 100,000 cycles and have almost no impact on performance.

In the context of SWAT recovery, the performance aspect of SafetyNet is certainly attractive. However, the on-chip buffers incur permanent cost to the system and grow with the recovery interval, which is the product of the length of the checkpoint interval and the number of stored checkpoints. If the detection latency of the SWAT detectors is close to 1,000,000 cycles or more, the CLB size will certainly exceed 512KB in order to avoid significant performance degradation. However, an on-chip buffer of this size bears a non-negligible cost even in modern mainstream systems, especially when this piece of hardware is used solely for ensuring reliability. Because of the potentially large area overhead incurred by the CLBs (permanent cost), we use the alternate scheme, ReVive, for our further investigation, as described below.

ReVive

Contemporary to SafetyNet, Prvulovic et al. proposed ReVive that slightly modifies the directory controller to (1) checkpoint the memory state by storing the undo logs in the main memory and (2) generate distributed parity of the memory (including the logs) to recover the system from permanent errors in one node [59]. To establish a globally consistent checkpoint, all nodes are synchronized through barriers to checkpoint the processor state and flush the dirty cache lines to memory. Because modern systems usually use ECC to handle errors in memory, we focus on the checkpointing aspect instead of the parity protection aspect of

ReVive.

In this scheme, before the start of a checkpoint interval, all cache lines are written back to the memory (i.e., no dirty cache lines in cache). As the execution continues, processors issue writes to different cache lines. Subsequently, the directory controller receives *upgrade* or *get-exclusive* requests and learns that a cache line is about to be modified for the first time since the last checkpoint. Hence, the controller saves the current version of the cache line in the undo log region of the memory. The modified memory state and the undo logs allow the system to be rolled back to the last checkpoint. Because the directory controller creates undo log entries by keeping track of the first writes to the memory blocks since the beginning of the interval, all processors flush the dirty cache lines to the memory (downgraded to *shared* state) before the start of the next interval. As all processors are synchronized periodically to take globally consistent checkpoints in ReVive, the synchronization overhead is always incurred. Further, first writes to any cache lines since the last checkpoint always result in misses, incurring additional latencies. To amortize these overheads and yield lower performance degradation due to checkpointing, the system can be configured to have longer checkpoint intervals. From the experimental results in the ReVive work, this method incurs an average performance overhead of 6% at 10ms checkpoint intervals [59].

One of the design goals of ReVive is to modify as little hardware as possible; only the directory controller is enhanced as a result. One may argue that the area overhead of ReVive is actually higher because a part of the memory is provisioned to store the undo logs, effectively reducing the amount of available memory to the running system. However, as shown in the experiments in [59], the undo logs only consume 2.5MB (the sizes of the logs for FFT and Radix are at this maximum) or less of memory per node per 10ms interval, making this overhead negligible when compared to modern systems equipped with gigabytes of memory.

Overall, as the two main design aspects of ReVive’s checkpointing scheme, minimizing hardware overhead and storing logs in memory, seem to align with the SWAT approach of keeping the common-case cost low, ReVive can likely be leveraged as the viable cost-effective SWAT recovery method. In our experiments, we investigate the applicability of ReVive as the SWAT recovery method.

6.4 Input/Output State Buffering and Recovery

As discussed in Section 6.1, full system recovery depends not only on rolling back to the pristine execution state, but also on proper handling of I/O events to contain faults. In particular, because output events are irreversible once committed, it is crucial to buffer output events until the system state is validated to be fault-free. As virtually every piece of software in the system interacts with I/O devices during its course of execution, this buffering mechanism is vital to ensure system recoverability.

In SafetyNet, it is suggested that I/O activities can be buffered during checkpointing. Hence, after recovery, the input events can be replayed to the system and the output events can be re-generated from the recovered system. However, the actual design and implementation of the buffering mechanisms was unknown at the time. Recently, researchers presented one output buffering method called ReViveI/O [50] for buffering disk and network events.

Using a software layer called pseudo device driver (PDD) that resides between the OS kernel and the device driver, ReViveI/O buffers all disk and network write activities so that potentially faulty events cannot propagate to the rest of the system (or the outside world). On the other hand, ReViveI/O does not have a mechanism for buffering input events because the targeted disk and network input events can be naturally replayed. For disk reads, ReViveI/O relies on the program to re-generate these read requests after a rollback. For reads of the network packets, the scheme relies on the retransmission feature in TCP/IP and lets the other machine resend the packets after a failure (a time-out mechanism) to receive acknowledgments.

In SWAT, as activated faults will manifest in software, they can cause faulty outputs to propagate to the rest of the system before they are detected. Hence, output buffering is particularly important in SWAT recovery. While ReViveI/O presents a feasible solution for disk and network activity, there are issues to be considered before integrating it with SWAT. First, the design of ReViveI/O involves modifying the internals of the OS, which is non-trivial as the complexity of modern operating systems continues to grow. Second, it is unclear how the method can be expanded and generalized to other kinds of I/O events. Third, because faults can manifest in software in SWAT systems, ReViveI/O's PDD software can be corrupted and send faulty outputs to the outside world before the error is detected. Given the stated reasons, we do not consider directly employing ReViveI/O in SWAT recovery. Nevertheless, this buffering scheme does present an effective approach for preventing fault propagation.

While ReViveI/O presents a software approach for output buffering, we realize this scheme by assuming that existing hardware can be enhanced slightly to achieve the same purpose. We further assume that this hardware module is highly reliable. This way, not unlike ReViveI/O, output events generated from the processor will be buffered in the hardware module until the checkpoint is validated to be correct. Subsequently, the buffered requests are sent to the devices in the system. If an error is detected, the requests buffered in the hardware module are discarded to prevent the fault from propagating to the outside world. For input events, we use the similar approach as ReViveI/O and rely on requests being regenerated during re-execution. Hence, we do not buffer input events for replay. For mainstream desktop or laptop computers, since all I/O events go through the northbridge (e.g., graphics card access, network card access, etc.), one possible implementation of this hardware module is to enhance the northbridge with this buffering capability.

6.5 Exploration of Checkpoint Recovery and I/O Buffering Methods

One shortcoming of many existing proposed checkpointing and I/O buffering schemes is the lack of quantitative results that show fault propagation can indeed thwart the error recovery process. That is understandable as most schemes are introduced as stand-alone solutions. Since we propose SWAT as a complete hardware reliability solution, it is of high importance to investigate how faults detected by the SWAT symptom-based detectors can be recovered in the system.

To study system recoverability, we explore the following configurations that make use of checkpointing and/or buffering techniques to recover faults that have been detected by SWAT’s symptom detectors.

- **Processor and partial memory state checkpointing.** As previously mentioned, the design of ReVive that stresses low area overhead aligns with the SWAT approach. Hence, we first explore this hardware-based processor-and-memory-level checkpointing scheme. In this configuration, to establish a checkpoint, the processor state is checkpointed with the use of shadow registers and the undo log is created for all memory blocks that have been modified by the processor since the checkpoint. We consider the memory state to be partially checkpointed because this method does not log the device-to-memory requests. (We discuss this issue in the next method.) From this scheme, we can understand whether a full system recovery can be achieved by focusing on handling the interactions between the

processor and the main memory. This method, therefore, serves as a baseline system to show the efficacy of processor-and-memory-level checkpointing in terms of system recoverability.

- **Processor and complete memory state checkpointing.** In the original ReVive work, the experiments were conducted on an architectural simulation framework that did not include simulation details of transactions to or from the devices (e.g., hard disk drive, network interface card, etc.). Therefore, the lack of a full system simulation environment prevented the experiments from capturing the interactions between the memory and the different devices in the system. However, in modern systems, devices often interact with memory to transfer data back and forth. Direct Memory Access (DMA) transfers, which allow large blocks of data to be transferred between the memory and the devices without the intervention of the processor, are prime examples of this type of interaction.

Building on the partial memory state checkpointing method, we added support to the checkpointing scheme to also log interactions between the devices and the memory. This method represents ReVive deployed in the realistic full system environment. In the event of a rollback recovery, this method patches the parts of memory that were previously modified by the devices, in addition to recovering the memory regions modified by the processor. Since all changes made to the memory state are logged, we call this scheme the complete memory state checkpointing.

- **Full system level checkpointing.** While the state of the processor and memory can be fully recovered with the above methods, a fault may still manage to corrupt the system by causing a faulty request to be sent to the device. One possible way to recover the corrupted system state is to apply checkpointing to the entire system. That is, the state of each device in the hardware system is also part of the checkpoint where the system can roll back to.

This method, although difficult to implement in practice, will allow us to understand the limit, if any, of checkpointing in terms of system recoverability. In other words, if all detected errors can be masked with this method, then full system recovery is attainable with system-wide checkpointing. However, if there exist detected cases where the system cannot be recovered, additional support beyond checkpointing for recovery will be needed.

- **Processor and complete memory state checkpointing with buffering of CPU-to-device write re-**

quests. While the above methods focus on checkpointing and aim to restore the pristine state of the system, a fault can still propagate outside of the system. For example, a bad request is sent to the network interface card as a result of the fault. Subsequently, this bad request turns out to be a part of the network packet to be sent to the other designated machine on the network. Only after this event, the fault is detected and the rollback recovery process is triggered. Unfortunately, at this point in time, the fault has become visible by the other machine and its effect is irreversible.

To mitigate situations like this, a buffering mechanism needs to be employed for holding the output request temporarily until the system state has been verified. As a result, we enhance the processor and complete memory state checkpointing with support for output event buffering. In this scheme, all CPU-to-device write requests are buffered until the next checkpoint. If an error detection occurs during a checkpoint interval, the buffered requests are discarded so that the potential faulty events do not propagate outside of the faulty system. As discussed in the last section, we use a hardware module to intercept and buffer the CPU-to-device requests.

Overall, with these proposed methods, we can view the SWAT recovery module to consist potentially of two essential sub-modules to enable full system recovery: (1) leveraging processor-and-memory level checkpointing for execution state restoration and (2) buffering CPU-to-device requests for preventing the irreversible effect of fault propagation to the outside world. In our experiment, we aim to find out the importance of both techniques in terms of system recoverability and their potential overheads.

6.6 Methodology

System reliability depends not only on how well a detection mechanism detects an error, but also on how capable the recovery mechanism is able to mask the detected errors. In particular, we want to understand whether employing only the checkpointing mechanism would suffice. Hence, in our experiments, we first investigate the system recoverability when the different recovery methods described in Section 6.5 are applied to the errors that are detected by the hardware-only detectors (Section 4.6). After that, we set up experiments to look into the overheads incurred by the recovery mechanism that potentially contains hardware checkpointing and output buffering. We describe these experiments in detail below.

6.6.1 System Recoverability

For systems that employ checkpointing for rollback recovery, system recoverability may depend not only on the integrity of architectural state (processor and memory state) but also on the proper handling of I/O events. In SWAT, because faults can manifest in the software before becoming detectable symptoms, preventing faults from propagating outside the system before detection is key to full recovery.

To create an environment where I/O activities are the norm, we focus on server workloads to gain full understanding of how SWAT ensures full recovery of the system. In particular, we investigate the following four methods described previously in Section 6.5. The following describes how these methods are implemented in our experiments. We apply these methods to only the server system in the two-system simulation environment (described in Section 4.4.1) while running the server workloads.

1. **Proc+ParMem: Processor and partial memory state checkpointing.** In our simulator based on GEMS, we implemented ReVive to take snapshots of the register state and to generate undo logs of the memory state as the execution progresses. For recovery, the processor state is rolled back and the memory state is restored through the undo logs.

In our experiments, we take a checkpoint at the beginning of the 10 million instructions of detailed microarchitectural simulation. This corresponds to a checkpoint interval of 10 million instructions. After a detection, a rollback recovery process is triggered and the application is then functionally simulated to completion using Simics.

2. **Proc+FullMem: Processor and complete memory state checkpointing.** In our experiments, we enhance the ReVive mechanism in GEMS to generate undo logs of memory blocks that are modified by the devices in the system. That is, if a device modifies any part of the memory during a checkpoint interval in our simulation, this mechanism will create an undo log entry for the pre-modified version of each memory block before the block is written to for the first time since the last checkpoint. The rollback recovery and replay are identical to Proc+ParMem.
3. **Full.Sys: Full system level checkpointing.** To implement this method, we rely on Simics to take a checkpoint of the state of the processor, memory, and different devices of the server system (the client system is not checkpointed) before the 10 million instructions of microarchitectural simulation. After

a detection, in rollback recovery, the state of the server system is restored from the Simics checkpoint. The replay process is the same as Proc+ParMem.

4. **Proc+FullMem+Buffer_Output: Processor and complete memory state checkpointing with buffering of CPU-to-device write requests.** In our simulation, we employed the module described in Proc+FullMem for checkpointing the state of the processor and memory. To buffer output requests, we implemented a Simics module for storing the processor-to-device write requests and sending the requests out to the devices at the next checkpoint. We did not limit the size of this buffer so that we can probe the potential storage overhead needed to implement this buffering mechanism. If a rollback is triggered due to a detection, the stored requests are discarded because they are suspected to be faulty. After the rollback, the replay process is the same as Proc+ParMem.

To quantitatively study system recoverability, we apply the above methods on the fault injection campaign performed on the server workloads as described in Section 4.4.2. After the injected faults are detected by the SWAT detection mechanism, we assume that the diagnosis process correctly identifies the faults and the appropriate repair action is subsequently taken. Hence, the rollback recovery process is triggered post-detection. Then, we functionally simulate the restored state until the application completes or a symptom occurs. We note that this re-execution happens on the fault-free hardware as diagnosis and repair has already been performed.

To determine recoverability, we compare the output of the completed application to that of the fault-free run. If the outputs are identical, we consider the recovery was successful. If the outputs are different, we consider the system not properly recovered and classify the case as a silent data corruption (SDC). Further, if the system ends up hanging/crashing, we consider such cases as detected unrecoverable errors (DUE).

For each recovery method described above, we define system recoverability as the percentage of detected cases that yield identical outputs as the fault-free execution after the recovery process (i.e., the error effect is completely masked).

Achieving high system recoverability is certainly the major goal of any recovery method. However, implementing the method does incur costs in area, power, and performance. Therefore, in the following, we describe the experiments used for investigating the overheads of the potential recovery scheme.

Base Processor Parameters	
Processor Type	In-Order 1-wide
Clock speed	2GHz
Number of Cores on chip	16
Base Memory Hierarchy Parameters	
Data L1/Instruction L1	16KB each
L1 hit latency	1 cycle
L2 (Shared and Unified)	256KB to 2048KB
L2 hit/miss latency	6/80 cycles
Base Cache Coherence Protocol (between L1 and L2)	MOESI
Memory Size	512MB
Memory Consistency Model	Sequential Consistency

Table 6.2: Parameters of the simulated multicore system.

6.6.2 Performance Overhead of Hardware Checkpointing

Hardware checkpointing is essential for restoring the pristine execution state in SWAT. In particular, we focus on ReVive since it incurs low hardware overhead. On the other hand, the original ReVive work shows that the scheme incurs some performance overhead during fault-free operation. Because the original work focuses on one system configuration and one checkpoint interval, it is unclear how this performance overhead changes with different system configurations and checkpoint intervals. Therefore, we set out to investigate the performance overhead of ReVive for different system settings to help us find the optimal design parameters. Here, we emphasize that the optimal design parameter for checkpointing may not be optimal for the entire recovery scheme. For example, if SWAT recovery were to include output buffering, choosing a longer checkpoint interval will likely reduce the performance overhead of ReVive but may require the buffering mechanism to have a larger storage buffer, incurring higher area overhead.

To gain insight into the fault-free cost of checkpointing, we evaluate the performance overhead of ReVive on a 16-core system with varying cache sizes. Table 6.2 shows the system for this study. To take advantage of all the cores, we run four parallel applications from the SPLASH benchmark suite (FFT, LU, Ocean, and Radix), including the ones that give high performance overhead in ReVive [59].

We use the Ruby memory system simulator in the Wisconsin GEMS simulator along with Virtutech Simics to model a full system environment running SPLASH applications on OpenSolaris. (ReVive was simulated on a microarchitectural simulator that does not simulate the OS.) The main difference between

this setup and the base setup described in Section 4.4.1 is that we do not use the Opal microarchitectural simulator in GEMS. While using Opal would add accuracy to our experiments, it also significantly lengthens the simulation time. As much of ReVive’s performance overhead actually comes from the increased traffic and latency in the memory system [59], we use only Ruby to study the performance overhead of ReVive with simulations that span tens of billions of instructions (this would take a long time if we were to use Opal).

The ReVive work focuses on checkpoints that are 10 million cycles long. To understand how the overhead of ReVive changes with checkpoint intervals, we vary the checkpoint interval from 500,000 cycles to 50 million cycles. We choose 50 million cycles at the high end to see whether the overhead reduces significantly with even longer intervals. We pick 500,000 cycles at the low end to observe how significant the performance overhead becomes with frequent checkpoints. This interval is also close to the 400,000-cycle ($4 \times 100,000$ cycles) validation latency in SafetyNet.

Our experiments also aim to show the impact of different system configurations. Hence, we vary the shared L2 cache size from 256KB to 2048KB. (The ReVive work only reports results for 128KB private L2 caches.) We use a very small 256KB L2 cache at the low end to capture how ReVive performs when cache misses are frequent. For the high end, we use a 2048KB L2 cache to match the total L2 cache size in ReVive.

To quantify the performance overhead, we calculate the slowdown of a system equipped with ReVive when compared to the baseline system with the same system configuration (e.g., same cache sizes) but with no checkpointing.

6.6.3 Storage Overhead of Output Buffering

The sphere of recoverability defines the system components that are recoverable by the recovery scheme. At the boundary of this sphere, events need to be buffered before the state within the sphere is validated. After the state is validated (e.g., no error detection), the events are released to their destinations. Because we use processor and memory state checkpointing, an output event buffering mechanism, if necessary, should be in place to prevent faulty events from propagating beyond the processor and memory to the system devices.

To determine the potential I/O events that need to be handled by SWAT recovery, we use the Simics full-system simulator to observe the interactions between the CPU and devices, and the memory and devices. We created a module in Simics (much like the Wisconsin GEMS simulator) to intercept all read/write requests

issued by the processor to the devices (e.g., the network interface card) and all read/write requests issued by the devices to the main memory (e.g., DMA transfer from disk to memory).

While it has been previously shown that server workloads have higher OS activity than SPEC workloads (Chapter 4), we do not know how they compare in terms of I/O activity quantitatively. Intuitively, server workloads would have more I/O traffic as they often communicate with the disk and network. In our experiments, we compare our two server applications, Apache and SSH daemon, against the SPEC applications. For SPEC workloads, we show the results for mcf and parser because they have the highest I/O activity.

In our experiments, we aim to observe the I/O activity throughout the lifetime of the application execution in order to fully understand the maximum requirement for the buffer storage. Ideally, we would like to have a modern processor that runs at a high clock rate (e.g., 2GHz). However, the latency of the PCI bus would be relatively high when compared to the fast processor and we could not configure the PCI bus speed freely in Simics. Hence, we chose to use the default Simics configuration, which assumes a 75MHz processor. We note that the results could potentially be more conservative (e.g., the stated storage requirement may be higher).

We measured these I/O activities in terms of the amount of data that needs to be buffered, which includes both the address (64-bit) and the data (varies between 1 and 64 bytes) for accessing a specific device. We collect this information at different buffering intervals, ranging from 10,000 to 100 million instructions. As we already found that many detections happen within 100k instructions in Chapter 4, the low end of the chosen intervals caters to future detectors that have latencies under 10k instructions. At the high end, an interval of 100 million instructions aims to buffer events for the very few detections that take a long time to occur. Intuitively, if the buffering interval is shorter, the data buffer can be made smaller since the accumulated amount of data requests is less. However, the buffering interval is governed by the maximum detection latency supported by the checkpointing mechanism. By obtaining the necessary storage overhead for the buffering mechanism from our experiments, we can design the module in SWAT recovery responsible for handling I/O events accordingly.

6.7 Results

In this section, we first investigate the recoverability when the system has a fault. In particular, the experimental results help us determine whether checkpointing mechanisms alone are sufficient to ensure full recovery, or event buffering mechanisms are also needed. To the best of our knowledge, this is the first work that quantitatively evaluates the limitation, if any, of the processor-and-memory-level hardware checkpointing mechanism.

Based on the results obtained on system recoverability, we discuss the strategy for designing an effective SWAT recovery solution. After that, we evaluate the potential overheads of the subcomponents of SWAT recovery. At the end of this section, we discuss the potential recovery scheme for SWAT.

6.7.1 System Recoverability

As we found that SWAT symptom detectors are able to detect most of the injected permanent faults within 10 million instructions (Section 4.6.1), let us first investigate the system recoverability for the detection latency of 10 million instructions. This study aims to show if it is sufficient to take processor and memory state checkpoints or if full recovery requires additional mechanisms such as output event buffering. To investigate recoverability, we attempt to recover the permanent faults injected into the system running server workloads with the methods described in Section 6.6.1.

Figure 6.1 shows the recovery outcome when different methods are used for recovering the detected faults. The different bars represent the results under the different methods. The different stacks in each bar show the different outcomes of the detections under these different recovery methods. *Recovered* denotes the cases where the applications generated correct outputs after the rollback and re-execution. *DUE* represents the detected cases that end up leading to a detection after re-execution. (We observe system hangs in most of these cases.) If the re-executions of the detected cases generate different outputs than the correct output, we categorize them as *SDC*. The number at the top of each bar shows the recoverability as defined in Section 6.6.1.

Interestingly, from the *Proc+ParMem* bar in Figure 6.1, more than two-thirds (68.6%) of the detected errors *can* be fully recovered by simply rolling back the processor state and the memory blocks that are modified by the processor even with I/O-intensive server workloads running. This shows the checkpointing

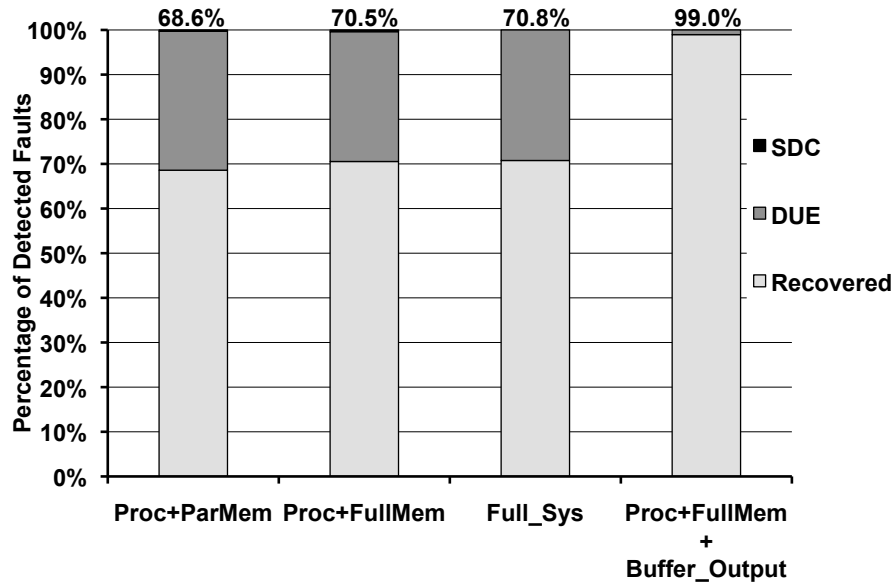


Figure 6.1: Recoverability of server workloads

method is very effective for the majority of the detected faults. Nevertheless, a significant number of the detections (31.1%) end up causing the system to crash/hang after rollback and replay. These cases essentially show the faults corrupt the system state and cannot be recovered by restoring the processor and memory state. Further, 0.3% of the detection results in incorrect output, indicating that faults do propagate outside of the system, become visible to the client system, and silently corrupt the data of the applications.

By also logging memory blocks that are modified by the devices in the system, *Proc+FullMem* is able to improve recoverability slightly to 70.5%. The difference in recoverability between this method and *Proc+ParMem* represents the percentage of faults that corrupt the memory blocks accessed by the devices but fail to be recovered by *Proc+ParMem*. Hence, if ReVive is to be deployed in real systems, the scheme has to handle not only interactions between the processor and memory but also those between the devices and memory. Nevertheless, the number of unrecoverable cases remains significant, with 29.1% causing system crash/hang and 0.4% producing faulty output. This shows that processor and memory state checkpointing alone is clearly insufficient to achieve high level of recoverability for these workloads.

Through checkpointing the entire system (including the state of the devices), one can determine whether faults can be fully recovered by simply restoring the state of the server hardware system. From Figure 6.1, *Full_Sys* improves the recoverability marginally, when compared to *Proc+FullMem*, to 70.8%. Though a

small number of faults can be recovered this way, these results show that merely checkpointing the entire state of the hardware system does not necessarily result in full system recovery. Specifically, by using *Full_Sys* to roll back the state of the processor, memory, and various devices, the system essentially has no memory of sending out any potentially faulty events to the outside world and is out-of-synch with the client system. As requests arrive at the server system, the server application and the OS detect many errors (unexpected inputs) and trigger the system to halt (to prevent critical errors from corrupting the system further). Essentially, the server system becomes the node in the distributed system that has the inconsistent state. These results therefore motivate the need for support for handling I/O activity.

Proc+FullMem+Buffer_Output enhances *Proc+FullMem* with buffering of CPU-to-device write requests. As shown in Figure 6.1, the number of recoverable faults increases significantly from 70.5% to 99.0%, with only 1.0% becoming DUEs. These results show that buffering the server system’s output events is highly effective in containing a detected fault and preserving the integrity of the system. This also suggests processor and memory state checkpointing cannot be deployed as the sole method for recovering detected faults in SWAT since our experiments show that faults do propagate to the rest of the system and become unrecoverable. In contrast to output buffering, input event buffering/replaying, although very important for achieving 100% recoverability, seems to play a lesser role in this context. This is because many input events are regenerated by the CPU (as pointed out by Nakano et al. in [50]) during re-execution. Given the high system recoverability, buffering output requests is vital to the integrity of the SWAT system, at least for the detections investigated here that have latencies of up to 10 million instructions.

6.7.2 Ensuring Full System Recovery in SWAT

From our experimental results for system recoverability, we see that both the checkpointing and output buffering mechanisms need to be involved for error recovery. We next discuss how the design parameters of these mechanisms impact system recoverability.

For SWAT and other systems that rely on checkpointing and buffering, the recoverability of a system depends on the *recovery interval* defined by the checkpointing mechanism, which subsequently impacts the output buffering mechanism. The recovery interval is defined as the maximum interval for rollback recovery, similar to the definition used in database systems [27]. In checkpointing, this interval is the product of the

checkpoint interval and the number of stored checkpoints. For example, if the checkpointing scheme has a checkpoint interval of 5 million cycles and is to keep three checkpoints, the recovery interval is 15 million cycles. In order to recover from an error detection, the detection latency has to be strictly shorter than this recovery interval. In other words, if the recovery interval can be made longer, more detections can be recovered and higher system recoverability can be achieved. However, if the recovery interval is longer, the buffering mechanism likely needs to store more events. This is because the system states are not validated during this interval. Hence, the buffering mechanism has to prevent potential faulty events from leaving the sphere of recoverability.

In SWAT, since the symptom detectors detect most faults within 10 million instructions, let us look at recovery schemes that handle this latency.

To ensure the pristine checkpoint is preserved, the recovery interval needs to be larger than 10 million instructions. One simple design choice for checkpointing is to use a checkpoint interval of 10 million instructions. In this case, the checkpointing mechanism needs to keep two checkpoints. As long as the fault can corrupt the memory state before it is detected, there can always be a scenario where the fault corrupts the state right before a checkpoint is taken but gets detected after the checkpoint. Hence, at least two stored checkpoints are always needed to ensure the restoration of the pristine state. Consequently, the recovery interval is $2 \times 10 = 20$ million instructions. Thus, the buffering mechanism needs to buffer outputs for 20 million instructions.

An alternative design is to take checkpoints more frequently, e.g., every 1 million instructions. In this design, 11 checkpoints are kept to make up a recovery interval of 11 million instructions. As a result, the buffering mechanism will need to buffer for 11 million instructions.

From these two examples, we can see that there is tension between the checkpointing and buffering mechanisms. For the same targeted detection latency, longer checkpoint intervals often yield longer recovery intervals. Therefore, more events need to be stored by the buffering mechanism, increasing the required storage overhead. To achieve shorter recovery intervals, checkpoints need to be taken more frequently. However, in ReVive, shorter checkpoint intervals generally result in more degradation in performance (discussed in Section 6.3.2).

Given these tradeoffs of the different design parameters in the checkpointing and buffering mechanisms,

we question whether a sweet spot can be sought to derive a recovery scheme that is low cost in area, performance, and power. Hence, in the following, we first look at the checkpointing overheads in the ReVive scheme for different checkpoint intervals. We then look at the output buffering overhead for different recovery intervals. After that, we draw conclusions for an overall recovery scheme for a given amount of recoverability.

6.7.3 Performance Overhead of Hardware Checkpointing

As the cost of DRAM continues to decrease due to device scaling, ReVive [59] may be a more attractive approach than SafetyNet [74] for implementing processor and memory state checkpointing. However, the performance overhead incurred by ReVive may negate the benefit of its low area cost. To investigate this overhead, we implemented the ReVive recovery mechanism in a multicore system and ran the benchmarks that were previously reported to incur high performance overhead in the original ReVive work.

Figures 6.2(a), (b), (c), and (d) show the slowdowns of four SPLASH applications caused by hardware checkpointing during fault-free execution. While the original ReVive work only shows the impact of ReVive for a fixed checkpointing interval (10 million cycles) on one multiprocessor system configuration (128 KB of private L2 cache), we vary the checkpointing interval from 500,000 to 50 million, shown on the x-axis, and the shared L2 cache size from 256 KB to 2048 KB, represented by the different lines, in our multicore system.

From the figures, we see that the checkpointing scheme incurs higher performance overhead as the checkpoint interval decreases for all four applications (as high as a 1.3x slowdown for Ocean). This is expected due to two effects: increased synchronization overhead and upgrade activity. For the former, while the time to synchronize different cores is more or less constant, this overhead becomes dominant as the checkpoint interval decreases. For the latter, the upgrade traffic increases and incurs overhead because the shorter checkpoint intervals force dirty cache lines to be downgraded (from *modified* to *shared*) more frequently. The result of this phenomenon is the increased number of write misses after the establishment of each checkpoint, degrading the performance noticeably. Here, we note that the reported overhead may be conservative since we assume a sequentially consistent system.¹ Nevertheless, the increased overhead for

¹Sequential consistency is the natural model in the GEMS simulation infrastructure, as also used for the work on the SafetyNet recovery evaluation [74].

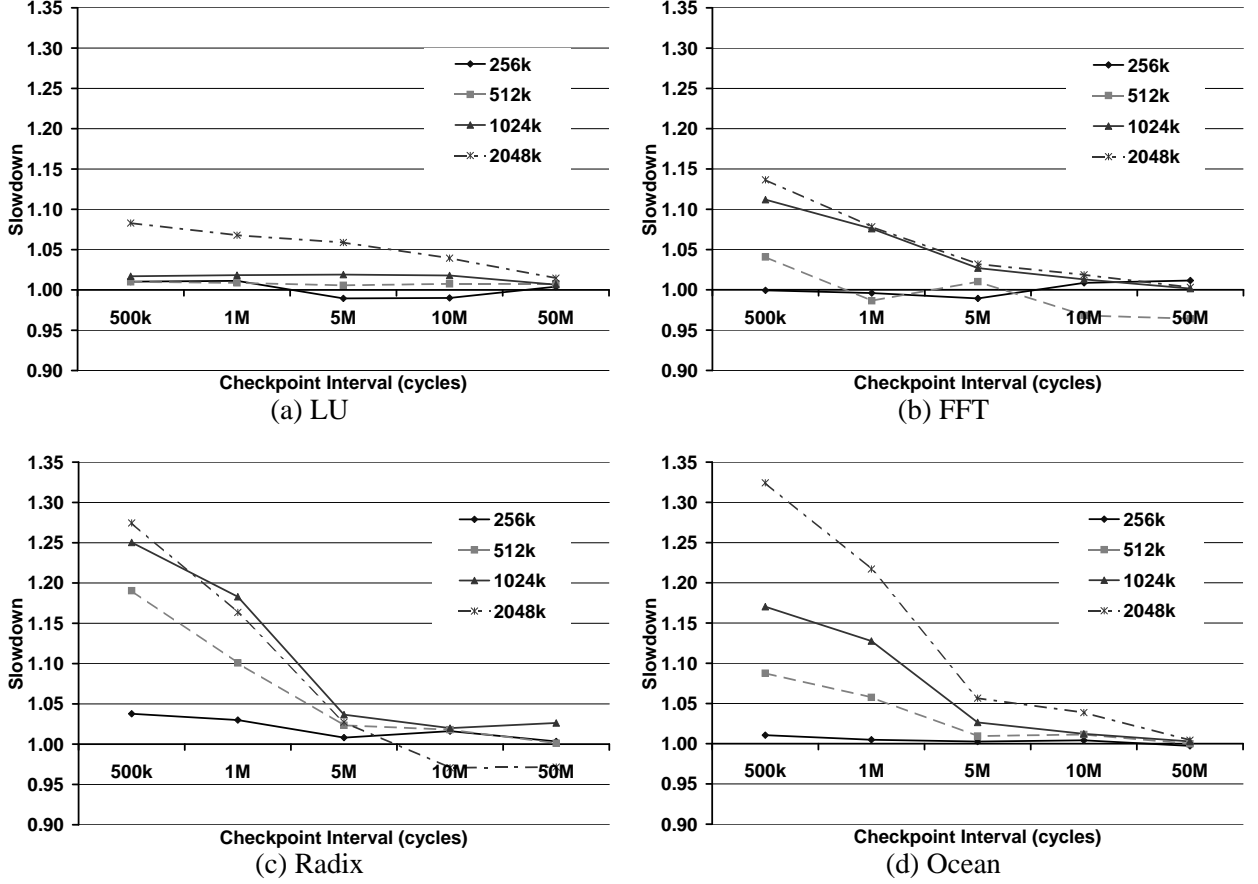


Figure 6.2: Slowdowns in fault-free execution of (a) LU, (b) FFT, (c) Radix, and (d) Ocean due to hardware checkpointing.

ReVive when using smaller checkpoint intervals is expected, as mentioned in [59].

Aside from the checkpoint interval, the cache size also impacts the efficiency of the checkpointing scheme. The ReVive work projects that *smaller* L2 caches would incur higher overhead when checkpointing and distributed parity are both enabled (Table 2 in [59]). Interestingly, when considering checkpointing alone, our results show that the checkpointing scheme incurs higher overhead as the L2 cache size is *larger*. In particular, the checkpointing scheme incurs no more than 4% overhead across all applications and checkpoint intervals in the system equipped with a 256 KB L2 cache. This is mainly caused by the low performance of the baseline system (without checkpointing). Since the working set does not fit in the L2 cache, the baseline system needs to replace cache lines more often due to increased capacity misses. This system behavior effectively reduces the impact of the checkpointing scheme because (1) many dirty lines might have

already been flushed during each checkpoint and (2) the checkpointing scheme hence has a similar cache line upgrade behavior as the baseline system.

From the ReVive work [59], three applications (FFT, Radix, and Ocean) were shown to incur over 13% fault-free performance overhead when distributed parity is enabled. When distributed parity is disabled (same as the scheme we implemented), these applications incur between 6% and 13% overhead. From our experiments, all applications incur less than 5% overhead for the same checkpoint interval (10 million cycles). This result is comparable to ReVive and the difference is likely caused by the slightly different system architectures and the difference in processor architectures. Namely, ReVive was deployed in a distributed shared memory system while the checkpointing scheme shown here is for a multicore system. Further, ReVive assumes the system to have aggressive 6-issue processors capable of handling multiple outstanding loads and stores. Here, we assume single-issue sequentially consistent processors. Despite these differences, the results shown here are very similar to what is reported in ReVive (e.g., a checkpoint interval of 10 million cycles).

From Figure 6.2, we see that using checkpoint intervals of 1 million cycles or shorter in this scheme incurs significant performance overhead, making it unattractive for SWAT. Nevertheless, ReVive is favorable for checkpoint intervals of 5 million cycles or more as the overhead stays below 6% for all four applications.

Since the detection latency we focus on is within 10 million instructions, checkpoint intervals of 5 and 10 million instructions seem to be the appropriate design choices. With shorter intervals, the performance impact would be too great. With longer intervals, since two checkpoints are always needed, the recovery interval would be much longer than 10 million instructions, putting pressure on the buffering mechanism. To understand this effect quantitatively, we next investigate the overhead of the output buffering mechanism.

6.7.4 Storage Overhead of Output Buffering

In Section 6.7.1, we found that output buffering is key to achieve high system recoverability. To further our understanding of the overheads involved for deploying a buffering mechanism, we investigate the I/O activities of Apache web server and SSH daemon, and compare them with two SPEC INT applications, mcf and parser. We measure these activities across different buffering intervals (which are the same as recovery intervals). For output requests, this interval is defined as the time period that write requests must be buffered

before they leave the sphere of recoverability and become visible to devices. For input requests, this interval is the time period that the requests need to be buffered for replay in case of a rollback recovery.

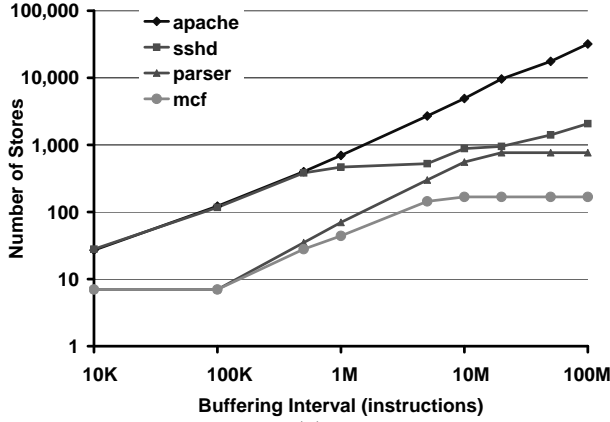
From Section 6.7.3, we found that the smallest checkpoint interval of ReVive that does not significantly degrade the performance of fault-free operation is 5 million instructions. With a maximum detection latency of 10 million instructions, the recovery interval is therefore three checkpoint intervals, i.e., 15 million instructions. Hence, the following discussion focuses on this interval length for buffering.

Figures 6.3(a), (b), (c), and (d) show the I/O activities in our applications across different buffering intervals. In particular, Figure 6.3(a) shows the maximum number of CPU-to-device stores, Figure 6.3(b) shows the maximum buffer size needed to hold these write requests (address and data), Figure 6.3(c) shows the maximum number of CPU-to-device loads, and Figure 6.3(d) shows the maximum buffer size needed to hold the read requests (address and data).

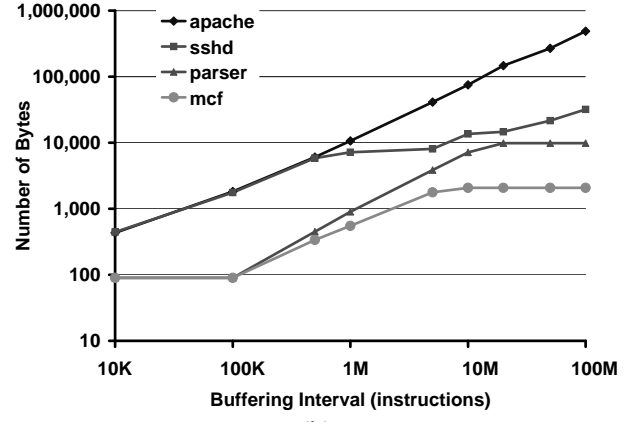
From Figure 6.3, as expected, longer buffering intervals result in higher number of requests and amount of data to be buffered. Of the four applications, Apache has the most I/O activities, followed by SSH daemon, then parser, then mcf.

When comparing Apache and parser (the SPEC application with higher I/O activities) at a buffering interval of 15 million instructions, Apache has 13 times the number of CPU-to-device stores (Figure 6.3(a)) and 15 times the amount of data to be buffered for these write requests (Figure 6.3(b)). For reads, the CPU-to-device bandwidth of Apache is 5 times that of parser. From these results, we see that not only the server applications have more OS activities, they also have high I/O bandwidth requirements than SPEC applications. Hence, this validates our use of server workloads for stressing the I/O buffering mechanism and helps us better understand system recoverability.

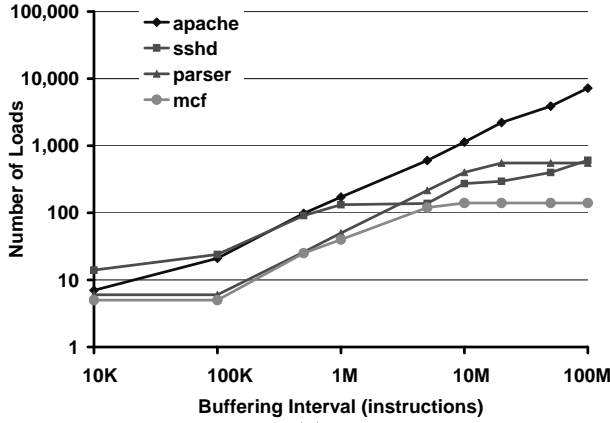
Besides the differences between SPEC and server workloads, there are variations between the two server applications as well. Apache and SSH daemon differ mainly when the buffering intervals become longer. Specifically, when the interval is 15 million instructions, the CPU-to-device write bandwidth (Figure 6.3(b)) of Apache is 10 times that of SSH daemon and the CPU-to-device read bandwidth (Figure 6.3(d)) is 8 times the bandwidth consumed by SSH daemon. With shorter intervals, the I/O events do not get accumulated as much when compared to longer intervals. Hence, the I/O activities between the workloads are similar. In longer intervals, the different behaviors of the applications start to show. By inspecting the CPU utilization



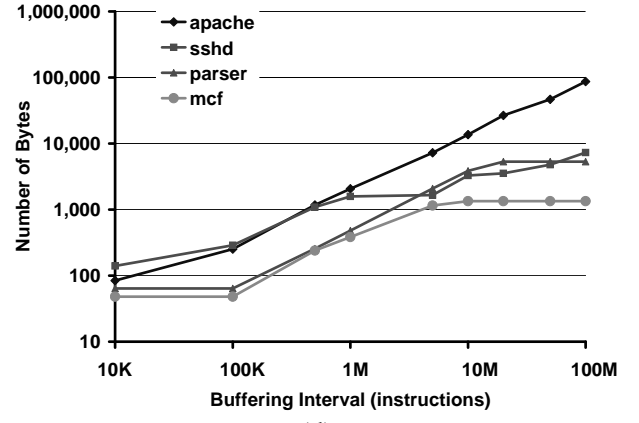
(a)



(b)



(c)



(d)

Figure 6.3: Maximum number and size of CPU-to-Device requests. (a) Maximum number of stores issued to the device by the CPU for varying buffering intervals. (b) Maximum buffer size for storing the CPU-to-Device write requests (address and data) for different buffering intervals.(c) Maximum number of loads issued to the device by the CPU for varying buffering intervals. (d) Maximum buffer size for storing the CPU-to-Device read requests (address and data) for different buffering intervals.

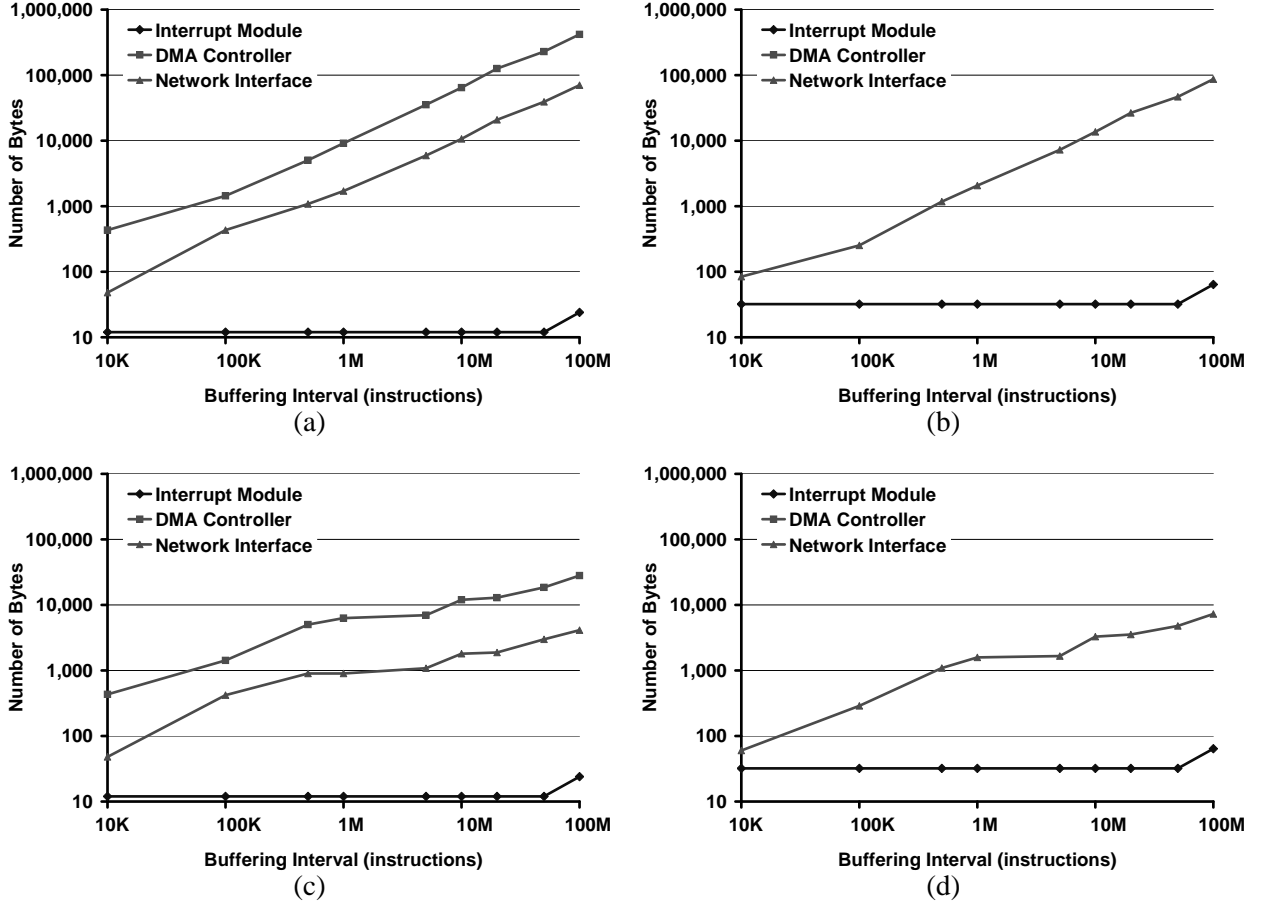


Figure 6.4: Interaction between CPU and specific devices in Apache and SSH daemon. (a) and (b) shows the maximum buffer size for storing device-specific writes and reads, respectively, in Apache. (c) and (d) shows the maximum buffer size for storing device-specific writes and reads, respectively, in SSH daemon.

of the system, we find that SSH daemon is more compute-intensive than Apache because of the encryption and decryption operations. As a result, the CPU is busy computing when running SSH daemon and generates relatively less I/O traffic than Apache.

To understand the sources of these I/O activities, we also look into the device-specific requests in Figures 6.4(a), (b), (c), and (d). Figures 6.4(a) and (b) show the device-specific storage requirements for CPU-to-device write and read requests, respectively, when Apache is running. Figures 6.4(c) and (d) show similar data when SSH daemon is running.

From Figures 6.4(a) and (b), the CPU reads 27KB of data and writes 21KB of data from and to the network interface card, respectively, in the 15 million instructions buffering interval for Apache. SSH daemon,

on the other hand, has read and write traffic of under 4KB from and to the network card as shown in Figures 6.4(c) and (d). As both Apache and SSH daemon are server applications, the majority of CPU reads and a large portion of CPU writes come from the network interface card, indicating the receiving and sending of packets to and from the connected clients. As mentioned earlier, SSH daemon is more compute intensive because of the encryption and decryption functions, the I/O traffic is relatively mild when compared to Apache.

The CPU-to-device write requests are dominated by the DMA controller at the host-to-PCI bridge. From Figure 6.4(a), at an interval of 15 million instructions, 126KB of data needs to be buffered for Apache. Figure 6.4(c), however, shows that only 13KB of data needs to be buffered for SSH daemon. These write activities are mainly for setting up the DMA transfers to move data between the network card and the memory. The rest of the I/O activity belongs to the PCI device that is responsible for delivering interrupts from the network device to the CPU.

As the server applications are under load, there is a significant amount of I/O activity within the system. Consequently, I/O buffering mechanisms with sufficient storage must be in place to prevent faults from propagating to the devices and the outside world. Once the effect of the fault becomes visible to the rest of the system, full system recovery is thwarted. For example, from Figure 6.3(b), we see that a 110KB storage is needed to sufficiently buffer output requests for Apache at a recovery interval of 15 million instructions to attain the high recoverability reported in Section 6.7.1. While this overhead may not seem significant, we note that the storage is dedicated only for reliability and improves neither power nor performance. Further, when compared to the near-zero hardware cost SWAT detectors, this mechanism is relatively costly. One potential cost reduction approach is to offload this buffering overhead to individual devices. As shown in Figure 6.4, a 20KB storage can be added on the network interface card to buffer write requests if the main goal of the buffering mechanism is not to send out bad network packets. Either way, as long as the executing software interacts with other devices in the SWAT system, buffering mechanisms are needed to stop fault propagation that affects full system recovery.

Depending on the reliability needs and the available budget, different buffering intervals can be chosen. As we target a checkpoint interval of at least 5 million instructions with the corresponding recovery interval of 15 million instructions, we discuss the possible overall recovery scheme in the following section.

6.7.5 Overall Recovery Scheme

With results presented in Sections 6.7.3 and 6.7.4, we now discuss the possible recovery scheme for SWAT. Specifically, since the processor-and-memory checkpointing and output buffering mechanisms are required to fully recover the system in most of the cases (Section 6.7.1), we focus on the possible overhead of this scheme.

Handling a Maximum Detection Latency of 10 Million Instructions

We start by looking at the recovery scheme for handling detection latencies that are within 10 million instructions. As discussed in Section 6.7.2, the recovery interval therefore needs to be larger than 10 million instructions.

One straightforward design choice is to pick a checkpoint interval of 10 million instructions. This way, the ReVive checkpointing scheme incurs a low performance overhead of 4% or less (with 2MB L2 from Figure 6.2). As we need to keep two checkpoints, the recovery interval is 20 million instructions. As a result, the buffering mechanism needs to have a 150KB storage (Figure 6.3(b)) for a buffering interval of 20 million instructions. In this design, while we chose the checkpoint interval that incurs low performance overhead, the storage overhead of the buffering mechanism is significant because of the long buffering interval.

Alternatively, to relieve pressure on the buffering mechanism, we can select a shorter checkpoint interval of 5 million instructions. From Figure 6.2, the performance overhead is 6% or less, a slight increase from the design above. With a maximum detection latency of 10 million instructions, the recovery interval needs to be three checkpoint intervals, or 15 million instructions. Thus, from Figure 6.3(b), the maximum output buffer storage needed is about 110KB (for Apache). Compared with the last design, the size of the buffer storage is reduced by 40KB while there is a slight increase of performance overhead for the checkpointing scheme. Nevertheless, with a buffering mechanism requiring over 100KB of storage, the area cost is still somewhat significant, especially when comparing to the very low cost SWAT symptom-based detection mechanism.

Overall, given these two potential configurations for the recovery scheme, we find the overheads in performance and area have room for improvement. In particular, while the performance overhead of the checkpointing mechanism may be sufficiently low for some systems, the large storage needed for the buffering mechanism, while not exorbitant, is larger than the very low cost always-on detection mechanism of

SWAT. Although we could not find a cost-effective recovery scheme at this point, we believe that this analysis prompts future research to focus on even lower cost checkpointing and buffering mechanisms.

Handling a Maximum Detection Latency of 100,000 Instructions

While the above two schemes aim to handle a maximum detection latency of 10 million instructions, future generations of SWAT may be able to improve this latency significantly. From our results in Section 4.6.3, close to 90% of the detections occur within 100,000 instructions. Assuming this to be the maximum latency for future SWAT detectors, the checkpointing mechanism needs to have a checkpoint interval of 100,000 or fewer instructions to minimize the recovery interval. This short recovery interval will drastically reduce the output buffering overhead to 2KB as shown in Figure 6.3(b) (100k buffering interval for Apache). However, from Figure 6.2, the ReVive scheme will certainly degrade the performance too much to be useful. One alternative is to use SafetyNet as the checkpointing scheme. As discussed before, the large CLBs (512KB) needed would incur too much area overhead.

Overall, when considering the shorter detection latency of 100,000 instructions, the buffering overhead is likely to be affordable. Nevertheless, there lacks an efficient checkpointing scheme that incurs low overhead in both area and performance. Therefore, a new low-overhead checkpointing scheme will need to be developed to achieve this short recovery interval.

Implementing Output Buffering

Besides storage overhead, one design issue for the output buffering mechanism is how it should operate. Because all CPU-to-device communications could potentially have harmful effects, the buffering mechanism needs to intercept the requests, store these requests, and release the requests (outside of the sphere of recoverability) when the system is validated to be correct. In modern PC systems, an ample location for this mechanism may be the northbridge, which handles communications among the processor, the memory, the video card, and the southbridge (responsible for communicating with other peripherals in the system).² To fulfill the storage requirements, SRAM modules (which continue to get larger and less expensive) may be

²For current and future systems that integrate memory and I/O controllers on-chip, designers may choose to integrate the buffering mechanism on-chip as well if the budget allows. Otherwise, the buffering module can still reside at the northbridge, as long as the datapath used by all output events are covered to ensure fault containment.

integrated with the northbridge to handle the buffering.

Alternatively, as the lowest level of cache continues to grow in size, this buffering mechanism can potentially reside on-chip. In this scheme, because the datapath between the processor and the I/O ports is usually separated from the datapath of the cache hierarchy, additional circuitry is used to re-route the I/O requests towards the cache that is reconfigurable for storing these requests. When the requests can be released, they are transferred off-chip from the cache. The disadvantage of this approach, however, is the added design complexity in the memory subsystem and the increased on-chip area cost.

If the detection latency is short enough (e.g., less than 10,000 instructions), these CPU-to-device write requests can potentially be kept in the store buffer. With this approach, very lightweight checkpointing mechanisms that can take frequent checkpoints at very low performance and area overheads are required. Another potential strategy is to leverage existing techniques introduced in transactional memory systems, where the cache is versioned to keep track of speculative states.

Overall, as we explored the possible implementations of the recovery scheme, we found that existing strategies are quite expensive. (We note that this is the case with many of the recent checkpoint/replay mechanisms proposed in the literature, not just SWAT. We, however, quantified this for the first time.) Therefore, a search for a new very low cost hardware checkpointing mechanism is required for building an effective SWAT recovery scheme.

6.8 Summary and Discussion

In this chapter, we explore the design of the possible recovery scheme for SWAT. Given the detection latency of the SWAT symptom monitors is relatively short, we focus on hardware checkpointing schemes that are capable of restoring the state of the processor and memory. In this context, our discussion assumes the sphere of the recoverability to include both the processor and memory. Since faults in SWAT can potentially propagate and cause faulty events to be sent outside of the sphere of recoverability, we also discuss the use of I/O buffering mechanisms.

Through our study of system recoverability using I/O-intensive server workloads, we found that both the checkpointing mechanism, for restoring pristine execution state, and the output buffering mechanism, for preventing faults from propagating outside of the sphere of recoverability, are equally important to guarantee

high system recoverability.

As ReVive incurs a low area overhead when compared to another contemporary scheme, SafetyNet, we use ReVive as the checkpointing mechanism of the recovery module. Nevertheless, ReVive’s performance overhead may be of concern. We then investigate the performance degradation for a number of SPLASH parallel applications and found that ReVive only impacts performance slightly when the checkpoint interval is 5 million instructions or longer.

For output buffering, we look into the storage requirements needed for holding potentially faulty output events. Our results show that the size of the buffer storage is significant when the checkpoint interval is in the range of millions of instructions.

With these findings, we discuss the possible recovery schemes and find that existing recovery strategies are quite expensive. Hence, from our experiments that investigate the system recoverability and the overheads incurred by the checkpointing and the output buffering mechanisms, we see two important future directions. First, to improve recoverability and minimize the cost for I/O handling, the detection latencies need to be reduced. Second, if the detection latency can be reduced, then there is a need for a rollback recovery scheme that can take frequent checkpoints (e.g., every 100,000 instructions) while incurring minimal cost in area, power, and performance. In concurrent work with my colleagues, we have already made significant strides on the former, reducing latencies by orders of magnitude using better detectors and metrics (not reported here) [64]. At these latencies (roughly 10’s of thousands), it may be possible to consider recovery strategies that only checkpoint the processor state and buffer memory accesses until they are validated (e.g., transactional memory style implementations). We leave this for future work.

Chapter 7

SWAT-Sim: Fast and Accurate Simulation of Permanent Hardware Faults

In the previous chapters, the evaluations of the detection, diagnosis, and recovery components of SWAT have been through microarchitecture-level fault injections, which has been used for other studies as well [4, 17, 38, 81]. However, because hardware faults occur at the device level but not at the microarchitecture level, it is unclear whether these evaluations are accurate. To answer the above question, this chapter introduces a methodology we developed that enables accurate modeling of hardware faults at the microarchitecture level. Leveraging the hierarchical simulation paradigm, our fault simulation infrastructure, *SWAT-Sim*, is able to accurately model gate-level faults at speed comparable to microarchitectural simulations [34]. SWAT-Sim not only allows us to accurately evaluate the SWAT system, but it is also applicable for evaluating other reliability solutions proposed at abstraction levels higher than the microarchitecture level.

For the rest of the chapter, we first motivate the needs and challenges for deriving an efficient fault simulation methodology. Then, we describe the SWAT-Sim infrastructure in detail. After that, we use SWAT-Sim in our experiments to answer three key questions of microarchitecture-level fault modeling: (1) Are the existing microarchitecture-level fault models accurate in representing gate-level faults? (2) If these models are inaccurate, what are the reasons? (3) Is it possible to derive more accurate microarchitecture-level fault models without simulating the gate-level faults? At the end of the chapter, we discuss the potential value of SWAT-Sim added to the ongoing and future work in both SWAT and other research in reliability.

7.1 Background

As the hardware reliability problem is expected to be pervasive across the entire computing market, several microarchitecture-level (μ arch-level) solutions that tolerate hardware failures have been proposed recently [4, 13, 17, 36, 38, 45, 73, 81]. The primary evaluation mode for these proposals has been through

statistical fault injections in simulations either at the gate level [13, 45, 73] or the microarchitectural state elements (e.g., output latch of an ALU) [4, 17, 36, 38, 81]. While gate-level fault injections can accurately capture lower level faults, the long simulation time of these schemes prevents detailed evaluation of the propagation of gate-level faults through the hardware and into the software. On the other hand, the μ arch-level injections are fast and allow observing faults propagated to the software level. However, while latch-level injections may be appropriate for array elements within the processor, it is unclear whether modeling faults in combinational logic at the latch level (e.g., injecting a fault at the output latch of the FP unit to represent a fault in the logic), is accurate. While alternative FPGA-based emulations [30, 56, 63] offer higher speed and model gate-level faults with high fidelity, the limited observability and controllability gives less flexibility than software simulations. Hence, this work focuses on software simulation methods.

The lack of speed in the gate-level fault simulation paradigm and the possible lack of fault modeling fidelity in μ arch-level fault simulation prompt searching for a solution that can achieve the best of both worlds. To address this classic tradeoff between speed and accuracy, we apply the paradigm of hierarchical simulation, where different parts of the system are simulated at different abstraction levels so that required details are modeled only in the parts of interest, thus incurring reasonable performance overheads [3, 11, 12, 29, 47, 57]. The resulting hierarchical simulator, SWAT-Sim, addresses the following criteria for simulating the system-level effects of gate-level permanent hardware faults.

In the context of fault tolerance, hierarchical simulations have been used to study transient faults in the processor by using a hierarchy of RTL and lower-level simulators [12, 47]. Since these simulators were used to study transients, they invoke the lower-level simulator just once to capture the effect of the fault, following which simulation happens only in the higher level. Other work has used hierarchical simulations to generate fault dictionaries that capture the manifestations from the lower level “off-line” and use them to propagate fault effects during high-level simulations [29]. This idea of fault dictionaries has also been used to study gate-level stuck-at faults in small structures, such as an adder [11]. However, fault dictionaries are specific to the fault model for which they are generated and cannot be used to simulate arbitrary fault models (the dictionary will have to be generated off-line for every such fault model); timing faults particularly present a challenge. Further, for faults in arbitrarily large structures, the growing sizes of inputs and faults make the dictionaries intractable, making them hard to use.

Our focus here is on the increasingly important permanent and intermittent faults [9, 84] and solutions for modeling them at the microarchitecture level or higher. In particular, successful solutions must address the following three critical aspects of fault simulation that prior work does not address in unison.

1. Simulation must be fast enough to capture how software would be affected by hardware faults.
2. Unlike transients, where the fault effect can be captured *once* and propagated to the higher abstraction level, permanent and intermittent faults have the characteristic that one activation of a fault could corrupt the software execution, which influences future activations of the same fault. This feedback mechanism between the hardware fault and the software must be faithfully simulated.
3. The simulator must be flexible enough to model different types of faults.

To meet the stated criteria, we propose a novel fault injection infrastructure, *SWAT-Sim*, that couples a microarchitecture-level simulator with a gate-level simulator and has the following properties.

1. To achieve speed close to a microarchitectural simulator and minimize overhead, SWAT-Sim only simulates the component of interest (in our case, the faulty component) at gate-level accuracy and invokes a gate-level simulation of the component *on-demand*.
2. To accurately capture the interaction between the hardware fault and the software, SWAT-Sim invokes the gate-level simulation repeatedly during runtime (interspersed with μ arch-level simulations); thus, if the software activates the gate-level fault, it would be corrupted and affects future activations of the same fault.
3. To allow fault modeling flexibility, SWAT-Sim employs a gate-level timing simulator where different timing faults can be modeled by changing the delay information within the faulty module.

These design choices of SWAT-Sim allow studying of the impact of gate-level permanent faults on software at speeds comparable to μ arch-level simulators. Further, since the fault simulation is performed while real-world software is executing, the effect of the fault is studied using functional vectors that represent realistic scenarios. SWAT-Sim thus has an advantage over other methods that use artificially generated test

vectors (e.g., functional vectors collected from a fault-free execution) to study the fault effect, as test vectors may not be representative of real-world faulty behavior.

In the following, we describe our SWAT-Sim infrastructure in detail and show how hardware faults can be modeled accurately and simulated efficiently with the SWAT-Sim approach.

7.2 The SWAT-Sim Infrastructure

SWAT-Sim is fundamentally a μ arch-level simulator that only simulates the faulty μ arch-level blocks, such as a faulty ALU or decoder, at the gate level. This greatly minimizes the gate-level simulation overhead.

7.2.1 Interfacing the Simulators

In SWAT-Sim, a gate-level Verilog module of the faulty unit is simulated only when the unit is utilized by the μ arch-level simulator. The inputs to the μ arch-level unit are passed as stimuli to the gate-level simulator. When the gate-level simulation completes, the results are passed back to the μ arch-level simulator, which then continues execution.

This communication between the two simulators is achieved using UNIX named pipes. In the μ arch-level simulation, each time an instruction utilizing the faulty unit is encountered, the stimuli needed by the gate-level module are written to a dedicated stimuli pipe. After the gate-level simulation completes, the computed data is written to a dedicated response pipe from where the μ arch-level simulator can read the response.

While the μ arch-level simulator can access the named pipes like files, the gate-level simulator is enhanced with two system tasks, implemented using the Verilog Procedural Interface (VPI) [16], that handle accesses to/from the pipes: One collects signals from the stimuli pipe and the other writes the results to the response pipe. The stimuli and response (arguments of the two tasks) are tailored to the μ arch-level structures under fault injection.

Figure 7.1 compares how a single fault in a μ arch-level structure X is simulated in a purely μ arch-level simulator (Figure 7.1(a)) and in SWAT-Sim (Figure 7.1(b)).

In Figure 7.1(a), a single fault in X is modeled as a single-bit corruption at the output latch of X because the μ arch-level simulator lacks the gate-level details of X.

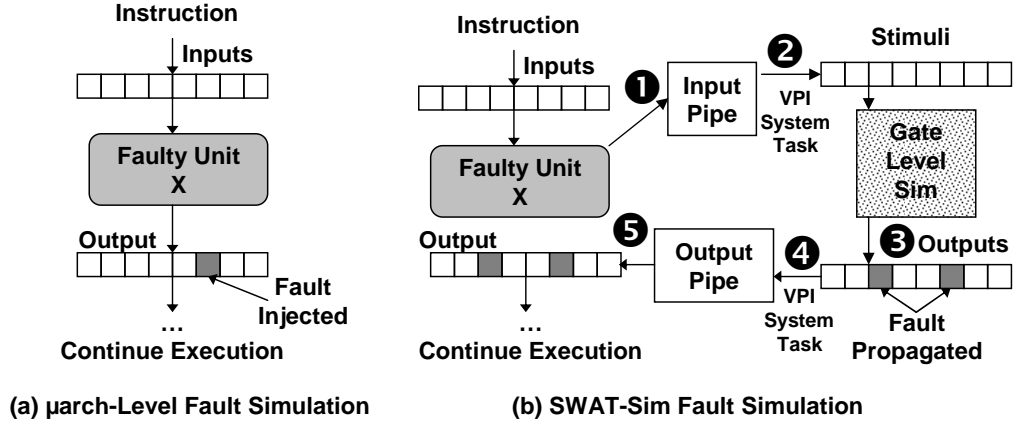


Figure 7.1: Comparison of how a faulty μ arch-level unit X is simulated by (a) a pure μ arch-level simulator and (b) by SWAT-Sim.

On the other hand, at the gate-level, a single fault in X is modeled as a fault in a specific gate or net. Figure 7.1(b) shows the steps of how the SWAT-Sim hierarchical simulator simulates the effect of this fault. (1) An instruction in the μ arch-level simulator uses X. SWAT-Sim collects the relevant input vectors and sends them to the stimuli pipe. (2) The Verilog system task reads from the input pipe and sends the stimuli to the gate-level simulator. (3) The gate-level simulator feeds the stimuli to the faulty module and obtains the output after gate-level simulation. (4) The Verilog system task transfers the result from the gate-level simulator to the response pipe. (5) The μ arch-level simulator reads the result from the response pipe and continues simulation. In particular, the figure shows the effect of a single gate-level fault propagating into a multiple-bit corruption at the output latch. In contrast, the fault injected in pure μ arch-level simulation only results in a single-bit corruption (Figure 7.1(a)).

During fault simulation, as long as the faulty unit is rarely utilized in the processor, the speed of SWAT-Sim approaches that of a microarchitectural simulator. If the faulty unit is heavily exercised, SWAT-Sim would spend more time in gate-level simulation and run slower. Nevertheless, the gate-level simulation of the faulty unit in the processor will certainly be more efficient than the traditional gate-level simulation of the entire processor.

Besides speed, accurately simulating the behavior of persistent faults, such as permanent faults and intermittent faults, is of high importance. To this end, SWAT-Sim invokes the gate-level simulation *on-demand during runtime*. With this setup, SWAT-Sim is capable of capturing the following scenario. At

the beginning, the software execution activates a persistent hardware fault. Subsequently, the fault corrupts the software execution. Then, some time later, this corrupted execution activates the fault again and gets corrupted further. This process can repeat many times during one fault simulation run. Here, we note that this interaction between the software and the underlying hardware fault cannot be easily modeled if the microarchitectural simulator and the gate-level simulator are two separate entities. In SWAT-Sim, on the other hand, this characteristic of persistent faults is correctly captured.

7.2.2 Different Microarchitecture-Level Structures

Given the wide variety of structures within a modern processor and the differences in the abstraction levels between a typical μ arch-level simulator and its corresponding gate-level counterpart, several factors should be considered when performing such hierarchical simulations.

- **Simulating sequential logic:** Simulating combinational logic with single- or multi-cycle latency in SWAT-Sim is straightforward. As long as the outputs are read after the stipulated latency, the outputs are guaranteed to be correct for each invocation. Sequential logic, however, requires state to be maintained across invocations. In SWAT-Sim, since the gate-level simulator is invoked (and thus clocked) only when the unit is utilized, state is maintained across multiple invocations, resulting in accurate simulation of sequential circuits.
- **Handling gate-level signals that are not modeled at the μ arch level:** In some cases, due to abstract modeling in the μ arch simulators, not all signals modeled at the gate-level appear at the μ arch level. If the faulty component contains such signals, the μ arch-level simulator can be enhanced with those signals to help propagate faults in these paths, improving its accuracy. Even in the absence of these enhancements, SWAT-Sim would present a more accurate fault model than existing μ arch-level fault models.
- **Simulating large μ arch-level components that may result in large overheads:** Since the primary aim of SWAT-Sim is being able to study the propagation of gate-level faults to the system level, simulations must be carried out at reasonable speeds. The components we study in this chapter present overheads in simulation time of under 3x (discussed in Section 7.4.1), when compared to pure μ arch-

level simulations. However, if the overhead becomes exorbitant because the faulty module is too large, the module can be further partitioned so that only the faulty submodule is simulated at the gate level while the rest is simulated at the higher level. For example, [47] uses such an approach in a lower-level hierarchical simulator.

Overall, by effectively coupling the gate-level and μ arch-level simulators, SWAT-Sim is capable of simulating gate-level faults in different μ arch-level components, making it a useful tool for full-system fault propagation studies with gate-level accuracy.

7.3 Methodology

7.3.1 SWAT-Sim Environment

Since permanent faults are persistent and can propagate through the μ arch-level to affect the OS and application state, SWAT-Sim requires a full-system, a μ arch-level, and a gate-level timing simulator. Any set of such simulators may be interfaced for the purposes of fault propagation.

In our implementation, SWAT-Sim consists of three components – the Virtutech Simics full-system functional simulator [80], the Wisconsin GEMS processor and memory μ arch-level timing models [42], and the Cadence NC-Verilog gate-level simulator. We interfaced the Cadence NC-Verilog simulator with GEMS using system calls implemented in VPI as described in Section 7.2

For the gate-level modules, we obtained the RTL designs of the arithmetic and logic unit (ALU) and the address generation unit (AGEN) from the OpenSPARC T1 architecture [77] and built an RTL model of the SPARC V9 decoder based on the decoder in GEMS. The Decoder module decodes one 32-bit instruction word per cycle and generates the signals modeled by our μ arch-level simulator. The ALU module is capable of executing arithmetic (add, sub), logical (and, or, not, xor, and mov), and shift (shift-left and shift-right) instructions. The AGEN module computes the effective virtual address given the operand values of the memory (load/store) instruction. Using Synopsys Design Compiler, we synthesized these modules at 1GHz with the UMC 0.13 μ m standard cell library. Further, this synthesis tool also generates the SDF (Standard Delay Format) file that contains the delay information of each gate and wire within the synthesized gate-level module. The Cadence NC-Verilog simulator then performs gate-level timing simulations with information

provided in this file. For delay faults (described in Section 7.3.2), we modify the post-synthesis SDF file to incorporate added delays.

7.3.2 Fault Models

In our experiments, we injected faults according to the following fault models to study differences in system-level effects among faults injected at the μ arch level and the gate level. In all cases, we inject single bit (or single wire) faults.

Gate-level stuck-at fault model: The gate-level stuck-at fault model is a standard fault model applied in manufacturing testing. We inject both stuck-at-0 and stuck-at-1 faults in randomly chosen wires in the circuit.

Gate-level timing fault model: It has been shown that aging-related faults result in timing errors in the faulty gate, with increasing delay as the aging worsens [7]. Ideally, we would like to model this effect using transition fault models and path delay faults, with different amount of delays. Here, we experiment with two delay fault models: (1) We inject a one-clock-cycle delay into the faulty gate such that timing violations occur along all paths containing the gate when a transition occurs. (2) The faulty gate is injected with a half-clock-cycle delay, potentially causing a subset of the gate’s output cone to violate timing.

Microarchitecture-level stuck-at fault model: Due to the absence of more accurate fault models, stuck-at faults at the input/output latch of a faulty μ arch-level unit have been used to estimate the effect of gate-level faults (both stuck-at and timing-related faults). We adopt this fault model, injecting both stuck-at-0 and stuck-at-1 faults at the input of the Decoder and the output latch of the ALU or AGEN.

7.3.3 Parameters of the Fault Injection

Our fault injection campaign is similar to the permanent fault injection experiments described in Section 4.4.2 except that faults are injected at 50, instead of 40, random points in each application (after initialization) and 3 structures, instead of 8, are studied. For the gate-level stuck-at and delay fault models, the 50 points in a structure are chosen from the 1853, 2641, and 757 wires of the synthesized gate-level representation of the Decoder, ALU, and AGEN, respectively. For the μ arch-level faults, these points are randomly chosen from the 32 bits of the input latch of the Decoder and from the 64 bits of the output latches of the ALU and AGEN. Further, since there are multiple decoders, ALUs and AGEN units in our superscalar processor, one of them

is chosen randomly for each injection. We also ensure that the samples are chosen so that gate-level stuck-at and delay faults are injected in the same set of wires to facilitate a fair comparison among the gate-level faults.

This gives us a total of 2000 simulations per fault model per structure ($4 \times 10 \times 50$). Each injection run whose fault is not masked is a Bernoulli trial for coverage (either detected or not). Further, since the injection experiments are independent of each other, this gives us a low maximum error of 1.1% for the reported coverage numbers, at a 95% confidence interval.

7.3.4 Studying System-Level Effects

A key objective of this study is to understand the differences, if any, in system-level manifestations of μ arch-level and gate-level faults within μ arch-level structures. For this purpose, we use the SWAT symptom-based detection scheme described in Section 4.5.1 because these detectors essentially capture how hardware faults manifest into the system level and software.

Given the injection outcomes (Figure 4.3), we study the differences between the various permanent fault models using detection coverage and detection latency of SWAT, as described in Section 4.5.3.

7.3.5 Limitations of the Evaluation

While SWAT-Sim is a flexible framework that is fast and accurate, it does have certain limitations. Here, we list some of the assumptions and limitations of our evaluation.

- SWAT-Sim assumes that a Verilog description of the module of interest is readily available for interfacing. This is true for the large fraction of the processor that is typically re-used from older tape-outs. However, for modules that are yet to be developed, neither SWAT-Sim nor pure gate-level simulators can be used to perform fault injection experiments. As these models start to become available, SWAT-Sim can be incrementally interfaced with them.
- Using SWAT-Sim, we study the propagation of gate-level faults in only three microarchitecture units (Decoder, ALU, and AGEN) as we could not find other Verilog modules close enough to the SPARC architecture modeled by the μ arch-level simulator (we used the in-order UltraSPARC T1 as our Verilog source and the out-of-order GEMS as our μ arch-level source).

- The timing information generated in the SDF file represents pre-layout timing, which does not reflect accurate post-layout timing for both gate delays and interconnect. By extracting this information using a place-and-route tool, the accuracy of our timing simulations, and thus our results, can be further improved.
- Although prior work has suggested other statistical delay models for timing faults (e.g., based on threshold voltage and temperature [55, 71]), we inject fixed and arbitrarily chosen delay that may or may not represent real-world failure modes. Integrating more accurate lower-level timing fault models in SWAT-Sim is a subject of our future work.

In spite of these assumptions and limitations, the results presented in this chapter demonstrate the importance of using hierarchical simulators, such as SWAT-Sim, to accurately model gate-level faults at the μ arch level.

7.4 Results

The hierarchical nature of SWAT-Sim allows us to achieve gate-level accuracy in fault modeling, at speeds comparable with μ arch-level simulators. We first summarize SWAT-Sim’s performance when compared to both the μ arch-level simulation and pure gate-level simulation (Section 7.4.1). We then use the SWAT-Sim simulator to first evaluate the accuracy of the previously used μ arch-level stuck-at fault models for representing gate-level faults (Section 7.4.2). Subsequently, we extensively analyze the reasons for the differences in the manifestations of gate-level faults from μ arch-level faults (Section 7.4.3). From this detailed analysis, we derive two candidate probabilistic μ arch level fault models for modeling gate-level stuck-at and delay faults (Section 7.4.4).

7.4.1 Performance Overhead

To understand whether SWAT-Sim’s hierarchical simulation infrastructure provides performance benefit, we profile a set of 40 fault-free runs for each structure and each fault model. We do not inject a fault in the desired faulty unit, but force the unit to be simulated at the gate level. To be conservative, we always use the most utilized unit for this purpose (e.g., ALU 0 for faulty ALU). For delay faults, we simulate the chosen unit

Unit	Fault Model	Maximum	Average
ALU	Gate Stuck-At	2.20	1.56
	Gate Delay	2.65	1.93
AGEN	Gate Stuck-At	1.59	1.26
	Gate Delay	1.89	1.35
Decoder	Gate Stuck-At	2.91	2.12
	Gate Delay	5.10	2.91

Table 7.1: Slowdowns of SWAT-Sim when compared to pure μ arch-level simulation.

with SDF timing annotation. Table 7.1 shows the maximum and average slowdowns of SWAT-Sim compared to pure μ arch-level simulation, when simulating the ALU, the AGEN, and the Decoder across different fault models.

Overall, the worst average-case slowdown of SWAT-Sim, compared to the μ arch-level simulation, is under 3x, which is an acceptable overhead considering SWAT-Sim’s ability to model gate-level faults. In particular, Table 7.1 shows that the Decoder incurs the most overhead, with average slowdowns of gate-level stuck-at and delay faults being 2.12x and 2.91x respectively. The average slowdowns of the ALU and the AGEN are under 2x. The maximum slowdowns observed for the ALU and the AGEN are under 2.7x and 2x, respectively while the overall maximum slowdown of 5.1x is measured for the Decoder. The Decoder incurs higher overhead than other units because it sits at the processor front-end and is more utilized than the ALU and the AGEN.

As expected, the delay fault simulations always incur higher overhead than the stuck-at fault simulations because simulating delay faults requires timing information which is more compute-intensive.

Since we do not have the corresponding gate-level model of the superscalar processor we simulate at the μ arch level, we derive a rough conservative estimation of the performance benefit as follows. Assume (conservatively) that we need to simulate a fault in a circuit that contains 4 times the number of gates and is utilized twice as often as the Decoder, the unit that incurs the most overhead. Assume that the full superscalar processor we wish to simulate has 25 million gates. Assuming SWAT-Sim’s worst-case slowdown is linear to the utilization and the size of the gate-level module and the baseline μ arch simulator simulates at the rate of 17k instr/sec (which is the measured average speed of our μ arch-level simulator), it would take SWAT-Sim $10M \text{ instr} \times \frac{4 \times 2 \times 5.1}{17k \text{ instr/sec}} = 6.7 \text{ hr}$ to simulate 10 million instructions in the worst case. On the other hand, conservatively assuming the gate-level simulator simulates 25M gates-

cycles/sec (more than 1300x the speed reported in [62]) and the execution has an IPC of 1, it would take $10M \text{ instr} \times \frac{25M \text{ gates}}{1 \text{ instr/cycle} \times 25M \text{ gates-cycles/sec}} = 2778 \text{ hr}$ to simulate 10 million instructions. SWAT-Sim thus achieves a 417x speedup over traditional gate-level simulation.

7.4.2 Accuracy of Microarchitecture-Level Fault Models

We next investigate the accuracy of μ arch-level fault models. If these fault models were accurate enough, then we can eliminate gate-level simulations entirely, thus eliminating the need for SWAT-Sim and its overhead. As mentioned in the last section, we focus mainly on the system-level effects of the fault models and use the SWAT detectors' coverage and latency as grounds for our comparisons.

Detection Coverage

Figure 7.2 compares the efficacy of the SWAT detectors in detecting different faults injected using different fault models into the ALU, the AGEN, and the Decoder. The bars represent the outcomes for the μ arch-level stuck-at-1 (μ arch s@1) and stuck-at-0 (μ arch s@0) models, the gate-level stuck-at-1 and stuck-at-0 models (Gate s@1 and Gate s@0, respectively), and the gate-level 1-cycle-delay and 0.5-cycle-delay models (Delay 1cyc and Delay 0.5cyc, respectively). Each bar shows the fraction of fault injections that are microarchitecturally masked (μ arch-Mask), architecturally masked (*Arch-Mask*), application-masked (*App-Mask*), detected within 10M instructions (*Detected*), detected but unrecoverable (*DUE*), and those that lead to silent data corruptions (*SDC*). The number on top of each bar represents the coverage.

Figure 7.2 shows that depending on the structure and the fault model, the μ arch-level fault model may or may not accurately capture the effect of gate-level faults, as indicated by the coverage. For the AGEN, the coverage of μ arch stuck-at faults is similar to that of the gate-level stuck-at and 1-cycle delay fault models (between 94% and 97%). However, the coverage of 0.5-cycle delay AGEN faults is noticeably lower (90%). For the Decoder and the ALU, the coverage for the μ arch-level stuck-at faults is near perfect (99+%) while the coverage of the gate-level stuck-at faults (94% for the ALU and between 96% and 98% for the Decoder) and the Decoder delay faults (95%) is slightly more pessimistic. In contrast, the coverage of the ALU delay faults is significantly lower (89% and 85% for 1-cycle and 0.5-cycle delay faults, respectively).¹

¹We found the coverage with SWAT-Sim improves significantly (from 89% to 94% for 0.5-cycle delay faults in ALU) when the undetected cases are run for 50M instructions, showing that SWAT's detectors remain effective at this longer latency (which is still

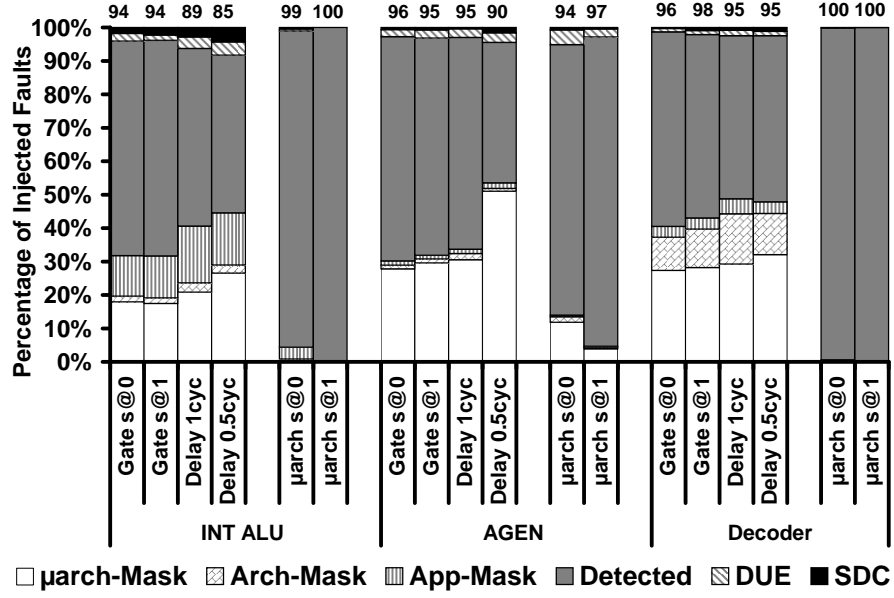


Figure 7.2: Efficacy of the SWAT fault detection scheme under different fault models for the ALU, AGEN, and Decoder. Depending on the fault model and the structure, the μ arch-level fault may or may not capture the system-level effects of gate-level faults accurately, as indicated by the differences in coverage.

The following analyzes the faults that do not result in detection in more detail.

Masking: A large source of discrepancy among the different fault models lies in the masking rate (μ arch-level, architectural, and application masking). The μ arch-level stuck-at fault models have very little masking of all three kinds (on an average, 0.3% for the Decoder, 2% for the ALU, and under 9% for the AGEN), while the gate-level fault models show a much higher rate of masking ($>30\%$ for all structures, with 0.5-cycle delay faults in the AGEN having the highest masking rate of 54%).

The masking rates of μ arch-level faults are low mainly because the faults are rarely μ arch-masked when compared to gate-level faults. As μ arch-level faults directly change the latch data, the only case where it does not result in a μ arch corruption (i.e., is μ arch-masked) is when the data does not activate the latch fault, e.g., correct data value of 0 masks a stuck-at-0 fault. At the gate level, there are two scenarios: (1) the fault at the gate is not activated, and (2) the fault is activated but does not propagate due to other signals in the circuit. Thus, the gate-level faults see much higher μ arch masking rates. Further, the μ arch-level faults are hardly masked at the application and architecture levels since they tend to perturb the data more severely and cause symptoms more easily than the gate-level faults.

recoverable [59], given appropriate support for checkpointing and I/O buffering is in place).

Interestingly, gate-level faults injected into the 3 structures exhibit different masking behaviors. All structures have high μ arch-level masking. However, architectural masking is significant only for the Decoder (25% to 31%) and application masking is substantial only for the ALU (35% to 42%).

Decoder faults are more likely to be masked at the architecture level than other structures. For these cases, we observe that the faults affect a subset of instructions of types that are sparingly used and corrupt only wrong-path instructions. Thus, even though the gate-level faults become microarchitecturally visible, they are not activated again after the pipeline flush and thus the faults are masked at the architecture level. For the ALU and AGEN, however, we see relatively few faults that get activated only by speculative instructions.

On the other hand, a significant number of ALU faults are masked by the application. This is likely due to the activated faults being logically masked. For example, suppose instruction $r1 \leftarrow r2 + r3$ uses the faulty ALU and the fault causes $r1$ to change from 1 to 2. If $r1$ is only used for the branch instruction *beq* $r1, 0, L$, the fault effect is masked by the application. This type of masking is relatively rare in other structures. Since it is more likely for Decoder faults to affect the program control flow and for AGEN faults to change the addresses of memory accesses, these faults, once activated, usually lead to detectable symptoms (i.e., not masked).

SDC: Similar to the overall coverage, the SDC rates (percentage of total injections that result in SDC events) are dependent on the type of fault and the structure in which the fault is injected. While the SDC rate is higher for gate-level faults than μ arch-level faults in the ALU (1.8%–4.4% vs. 0%–0.5%, respectively) and the Decoder (0.4%–1.2% vs. 0.1%–0.2%, respectively), the SDC rates of the AGEN faults are nearly identical (1.6% for 0.5-cycle delay faults and 0.5%–0.8% for others).

The SDC rates are high for the gate-level faults in the ALU because these faults are rarely activated and only perturb the data value slightly once activated. In contrast, the μ arch-level stuck-at faults are easily activated and less likely to cause SDCs.

The above differences in manifestations are largely governed by how the fault at the gate level becomes visible to the microarchitecture (activation rate, which latch bits are corrupted, etc.). In Section 7.4.3, we perform a more in-depth analysis to identify the reasons for the differences.

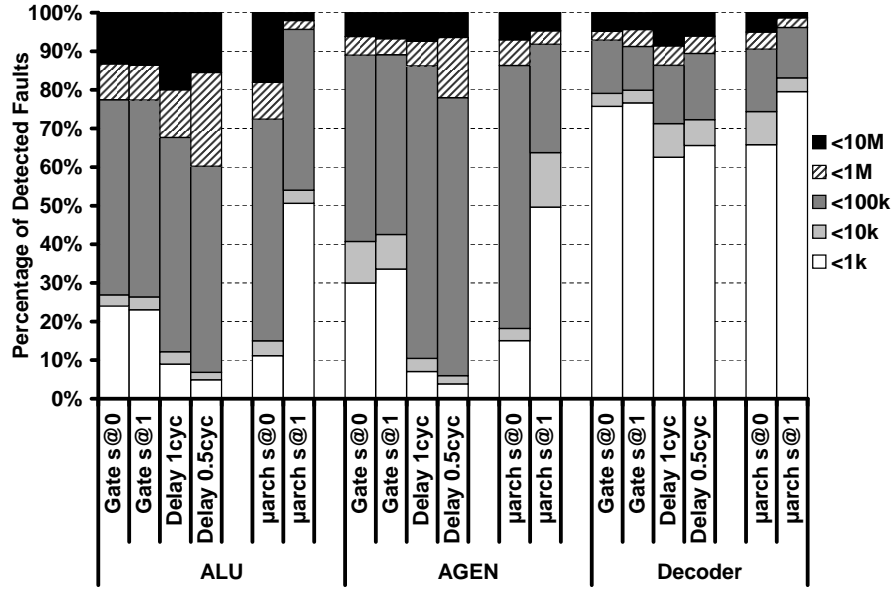


Figure 7.3: Latency of fault detection in terms of number of instructions executed from architectural state corruption to detection. The differences in the models impact recovery, which is primarily governed by these latencies.

Latency to Detection

Figure 7.3 gives the total number of instructions executed after the architectural state is corrupted, until the fault is detected, for each unit under each fault model. The detected faults are binned into different stacks of the bar based on their detection latencies (from 1,000 to 10 million instructions).

From Figure 7.3, we see that the percentage of detected faults that have latencies less than 10,000 instructions is different under different fault models for the three structures. Detections with these short latencies can potentially be recovered with light-weight hardware techniques (e.g., methods used in transactional memory systems). While the μ arch-level stuck-at-1 model shows that a larger fraction of cases have latencies under 10,000 instructions than gate-level stuck-at faults, the fraction of μ arch-level stuck-at-0 faults that have at most this latency is lower.

From these differences in system-level manifestations, we infer that μ arch-level stuck-at faults do not, in general, accurately represent gate-level stuck-at or delay faults. This motivates either building more accurate μ arch-level fault models, or in their absence, using the SWAT-Sim infrastructure to study the system-level effect of gate-level faults.

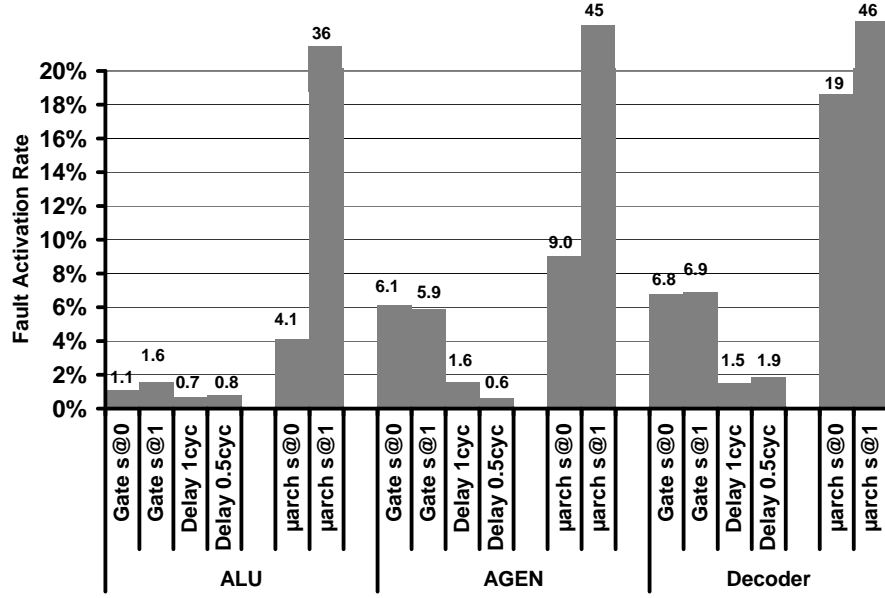


Figure 7.4: Mean fault activation rate for the different fault models as a percentage of the number of instructions.

7.4.3 Differences Between Fault Models

Before we attempt to derive a more accurate μ arch-level fault model than the existing ones, we investigate the fundamental reasons for the different behaviors of the μ arch-level and gate-level fault models. In the following sections, we try to understand the differences by comparing the fault activation rates and the data corruption patterns at the microarchitectural state across different fault models.

Fault Activation Rates

The fault activation rate of a given faulty run is defined as the percentage of instructions that get corrupted by the injected fault among all instructions that utilize the faulty unit. We collect the activation rates for all faulty runs that do not result in μ arch-masked, calculate the weighted arithmetic mean for each fault model, and present these numbers in Figure 7.4. Because the different runs execute different numbers of instructions, we weight the activation rate of each run by the total number of instructions executed by the faulty unit and calculate the weighted mean.

Figure 7.4 shows that the μ arch-level stuck-at faults present a higher activation rate than faults injected at the gate-level. For the ALU, the μ arch-level faults have a $>4\%$ activation rate, while the activation

rates of gate-level faults are at most 1.6%. For the AGEN, the corresponding numbers are $>9\%$ and $<7\%$, respectively. The Decoder faults tend to have higher activation rates than faults in other structures because decoders are utilized more; the Decoder μ arch-level faults have activation rates $>19\%$ while the rates of gate-level faults are $<7\%$. The activation rate for gate-level faults is lower because activating gate-level faults requires both excitation and propagation to the output latch, while the μ arch-level fault is directly injected into the latch. Additionally, the μ arch-level stuck-at-1 fault has a significantly higher activation rate than the other fault models (36%, 45%, and 47% for the ALU, the AGEN, and the Decoder, respectively). This high rate is caused by the biases in data values towards zero.

Further, we notice a difference in the activation rates between the gate-level stuck-at and delay faults, with the delay fault models exhibiting lower rates of activation for all structures. Less than 2% of instructions activate the 1-cycle delay faults and 0.5-cycle delay faults in all 3 structures, with the lowest average activation rate being 0.6% for 0.5-cycle delay faults in the AGEN. The lower average activation rate can be explained with the different excitation conditions for the two models. A stuck-at- X fault is excited when the signal at the faulty net is \overline{X} . Thus, if the probability of having a logic 1 at the faulty net is p , the probability of exciting the stuck-at-0 fault at that wire is p and that of exciting the stuck-at-1 fault is $(1-p)$. A delay fault, on the other hand, is active only if there is a transition at the faulty wire and hence the excitation probability is $p(1-p)$, which is always smaller than that of the stuck-at faults. This lower probability of excitation generally results in a lower average activation rate for gate-level delay faults. Further, while an activated 1-cycle delay fault causes all paths from the faulty net to the output latch to miss timing, a 0.5-cycle delay fault usually results in fewer errors observed at the output as it can be the case that some paths from the faulty net to the output do not violate timing.

Although the higher activation rates (Figure 7.4) of μ arch-level stuck-at faults result in higher coverage (Figure 7.2) for the ALU and Decoder, we do not find such a correlation for the AGEN. When comparing gate-level faults of the same structures, stuck-at faults have higher activation rates and result in slightly higher coverage than delay faults for the ALU and Decoder, but not for the AGEN. Nonetheless, higher activation rates do not necessarily drive the coverage up. Additionally, we find no direct correlation between activation rate and latency of detection. Thus, factors other than just activation rate need to be investigated if we are to succeed in deriving better μ arch-level fault models. We next look at how activated faults manifest at the

Bits	ALU				
	1	2	4	8	9+
μ arch	100.0%	0.0%	0.0%	0.0%	0.0%
Gates@1	91.1%	4.7%	1.2%	1.1%	1.9%
Gates@0	84.4%	4.6%	2.8%	1.1%	7.1%
Delay 1cyc	90.4%	3.9%	1.4%	1.1%	3.2%
Delay 0.5cyc	75.0%	5.8%	2.2%	3.9%	13.1%

(a)

Bits	AGEN				
	1	2	4	8	9+
μ arch	100.0%	0.0%	0.0%	0.0%	0.0%
Gates@1	87.1%	6.8%	5.0%	1.0%	0.1%
Gates@0	75.5%	8.4%	8.6%	7.4%	0.0%
Delay 1cyc	90.5%	4.1%	3.7%	1.5%	0.2%
Delay 0.5cyc	83.7%	7.9%	3.1%	2.4%	2.8%

(b)

Bits	Decoder				
	1	2	4	8	9+
μ arch	72.5%	0.2%	4.8%	8.9%	13.4%
Gates@1	66.1%	14.9%	10.5%	6.2%	2.3%
Gates@0	60.8%	22.3%	12.2%	2.6%	2.2%
Delay 1cyc	71.7%	11.1%	12.5%	1.7%	2.9%
Delay 0.5cyc	68.2%	12.8%	4.3%	2.7%	12.0%

(c)

Table 7.2: Percentage of bits incorrect at the output latch.

output latches (i.e., at the μ arch-level).

Corruption Pattern at the Microarchitectural State

While an activated μ arch-level fault corrupts only one bit in the microarchitectural state, an activated gate-level fault may corrupt multiple bits once it becomes visible in the microarchitectural state.

Table 7.2 shows the number of bits corrupted at the output latch (microarchitectural state) for different fault models for a fault in the ALU, the AGEN, and the Decoder. For each fault model, it shows the percentage of instructions that have different number of bits flipped at the output latch. The bits are binned on a log scale.

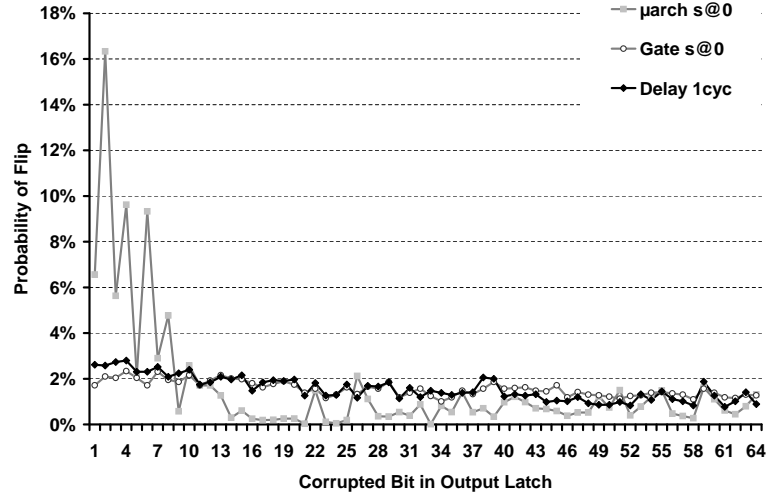


Figure 7.5: Probability of corrupting each bit of the ALU output latch, under μ arch-level s@0, gate level s@0, and gate level delay models.

Table 7.2 shows that the corruption patterns of μ arch-level faults for the ALU, AGEN, and Decoder are quite different from those of the gate-level faults. While μ arch-level ALU and AGEN faults are injected in the output latches and corrupt at most one bit, the corresponding gate-level faults, though usually corrupt one bit, can result in multi-bit corruptions (between 9% and 25% across the ALU and the AGEN). However, for μ arch-level Decoder faults, although faults are injected at the input latch, the resulting multi-bit corruptions turn out to be too aggressive (22% of corruptions for μ arch-level faults are 8+ bits while the corresponding numbers for gate-level faults are less than 15%). This is because the output cone of the input (output) latch of the faulty unit is too large (small) when compared to that of a gate-level fault and leads to aggressive (conservative) bit corruptions at the output latch.

To better understand how the microarchitectural state gets corrupted by the injected faults, we collect the probability that bit i was flipped, given an instruction activates the underlying fault. Figures 7.5 and 7.6 show the distribution of the probabilities of a given bit in the output latch (numbered from bit 0 to bit 63) to be faulty under μ arch-level stuck-at-0, gate-level stuck-at-0, and gate-level 1-cycle delay models for the ALU and the AGEN respectively. For brevity, we omit the μ arch-level stuck-at-1, gate-level stuck-at-1, and 0.5-cycle delay models.

From the figures, we see that the probabilities of bit-flips of the μ arch-level model are vastly different from the gate-level models. Further, the probability of flipping lower order bits is higher for μ arch-level faults

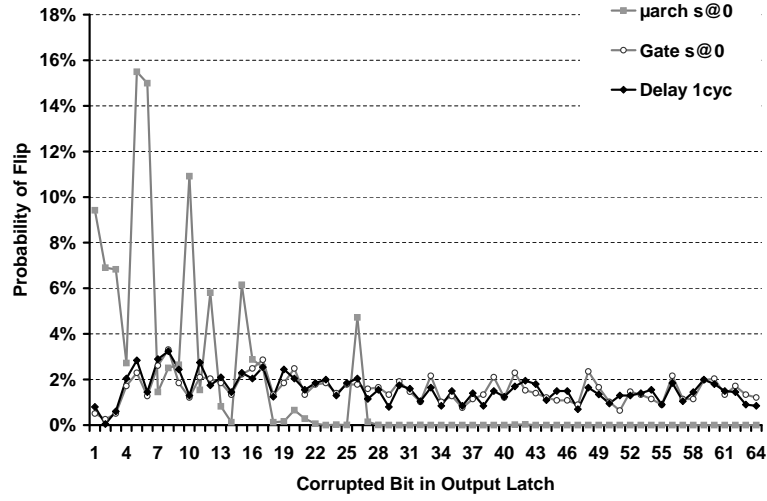


Figure 7.6: Probability of corrupting each bit of the AGEN output latch, under μ arch-level s@0, gate level s@0, and gate level delay models.

as the applications we use predominantly perform computations on the lower order 32-bits. The difference presented here is another source of discrepancy of the μ arch-level model to represent gate-level faults.

When comparing the two gate-level fault models, interestingly, both have very similar corruption patterns even though they differ in terms of coverage, detection latency, activation rate, and number of bit-flips. To investigate this phenomenon, we studied the differences between corruption patterns of the gate-level stuck-at and delay fault injected at the same net and made the following observation: delay faults generally yield more corruption patterns than the stuck-at-0 faults because they can cause the same bit to be corrupted in both directions, instead of a single direction in stuck-at-0 faults. While this higher number of corruption patterns may cause delay faults easier to be detected, we note that the average activation rate of delay faults is also lower than that of stuck-at faults, as explained in Section 7.4.3, making them harder to detect and causing longer detection latencies.

Overall, our analysis shows that the different activation rates and bit corruption patterns paint a clearer picture in explaining the differences in the coverage (Figure 7.2) and the detection latencies (Figure 7.3) between μ arch-level and gate-level faults. We found that higher activation rates of μ arch-level stuck-at-1 faults typically cause higher coverage (and lower detection latencies) than gate-level faults, but it is not a perfect correlation. In some cases, despite significant differences in activation rates, the coverage of gate-level and μ arch-level faults is quite close. This is because once activated, gate-level faults cause different

multi-bit corruption patterns. In some cases, these patterns are more intrusive than the μ arch-level fault corruptions, boosting the coverage of the gate-level faults despite their lower rate of activation. In other cases, the higher intrusiveness of the multi-bit corruptions is not enough to compensate for the very low activation rates – this is specifically the case for gate-level delay faults which see the lowest coverage numbers.

We see that such complex interactions have a push-and-pull effect in determining the system-level outcome of faults and conclude that simple μ arch-level stuck-at faults are inaccurate in several cases for modeling gate-level faults because they fail to (1) capture the system-level behavior, such as application-level masking, (2) induce different activation rates, and (3) accurately model μ arch-level multiple bit corruption patterns. (Nonetheless, these differences do not impact the quantitative results reported in previous chapters significantly and the previous qualitative results remain valid.) Therefore, any accurate μ arch-level fault model for gate-level faults must account for all these factors to accurately capture their behavior.

7.4.4 Probabilistic Microarchitecture-Level Fault Models

Given the inaccuracy of the μ arch-level stuck-at fault model, we investigate whether we can derive alternate μ arch-level fault models based on our analysis of the manifestation of the gate-level faults (both stuck-at and delay) at the microarchitecture level. Such a model would be invaluable for accurately simulating the effect of the fault at the μ arch-level, without invoking a gate-level simulator.

We investigated the behavior of the gate-level stuck-at and delay faults and found that each gate-level fault is activated differently and leads to different software-level outcomes. Hence, in our first-cut μ arch-level fault model, we develop probabilistic models on a per-run basis, i.e., a different probabilistic model for each injected gate-level fault. In particular, we profile each SWAT-Sim run and collect the probabilities of the number of bits flipped at the output latch, the patterns of the flips, and the directions of the flips. Based on the collected information, we then derive two probabilistic μ arch-level fault models, called the *P-model*, and the *PD-model*, respectively.

In the P-model, when an instruction uses the faulty unit, we decide on which bits to flip in the output latch based on the previously observed probabilities of the different number of bit-flips for this gate-level fault injection run (essentially using a table like Table 7.2, but built on a per-run basis). We then condition on this probability to decide on the pattern of the flip (similar to Figures 7.5 and 7.6 for different numbers of

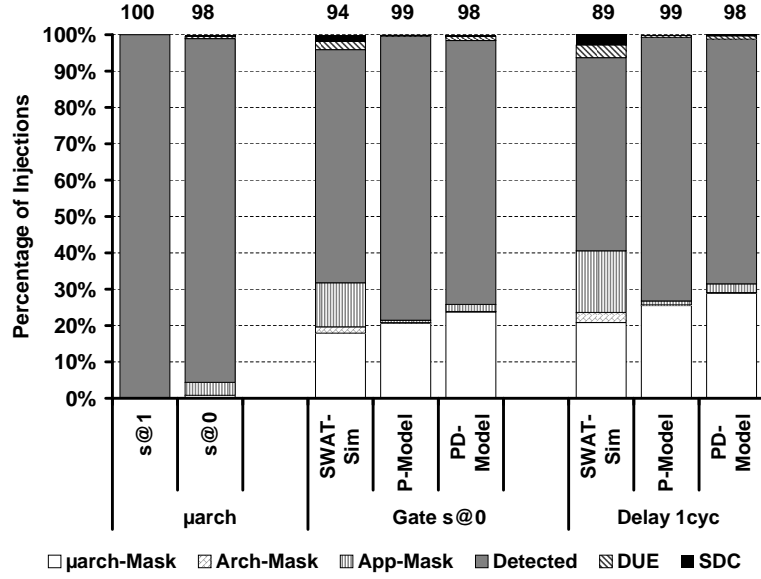


Figure 7.7: The accuracy of the derived P- and PD-models for modeling gate level faults in ALU, evaluated using the coverage of the SWAT detectors.

bit flips, but again on a per-run basis). All the bits indicated by this pattern are then flipped. This operation is done by XORing the output latch data with the collected bit-flip pattern. For example, if the data is 0110 and the corruption pattern is 0011, the corrupted output becomes $0110 \text{ XOR } 0011 = 0101$.

The PD-model refines the P-model by enforcing the direction of the bit-flips based on the profiling runs. That is, if the observed corruption pattern in the profiling run shows bit 3 of the output latch has a one-to-zero (zero-to-one) corruption, in the PD-model, this bit is corrupted only if it is an one (a zero). In our implementation, one word is used to represent the corruption pattern (same as the one used in the P-model) and another word is used to represent the corruption direction. Using the same example above, the PD-model has the corruption direction word of “— ↓” where ↓ means 1-to-0 corruption, the data is then changed from 0110 to 0100.

We developed the P-model and the PD-model for both the gate-level stuck-at-0 and 1-cycle delay faults for the ALU and the AGEN. Figures 7.7 and 7.8 show the ability of the P-model and the PD-model in mimicking the behavior of the corresponding gate-level fault models, evaluated using the coverage (similar to Figure 7.2). The number on top of each bar gives the coverage of the SWAT detectors for faults injected in that fault model. The results for gate-level stuck-at-1 and 0.5-cycle delay faults are not shown for the sake of clarity of the figures, and lead to similar conclusions as the other fault models.

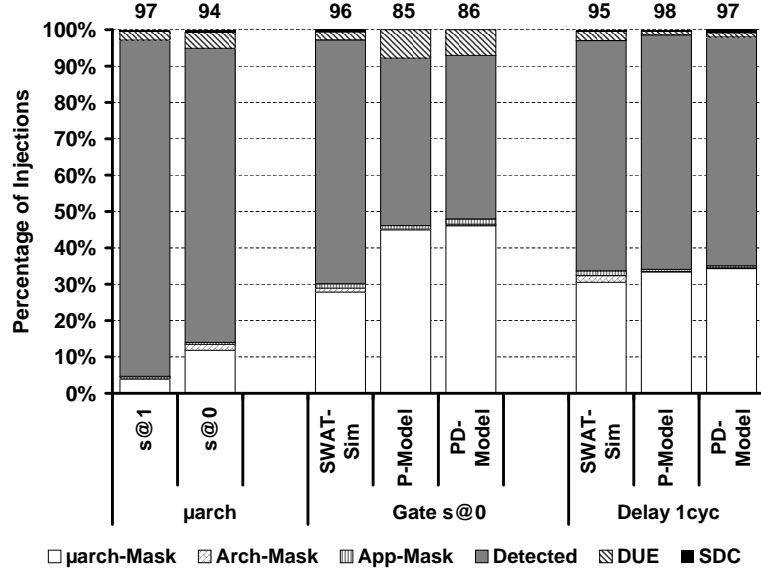


Figure 7.8: The accuracy of the derived P- and PD-models for modeling gate level faults in AGEN, evaluated using the coverage of the SWAT detectors.

From the figures, we see that both the P-model and the PD-model follow the μ arch-level masking effects of the gate-level faults more closely than the μ arch-level stuck-at faults. Nevertheless, the P- and PD-models for both gate-level stuck-at-0 and 1-cycle delay ALU faults are unable to capture the application-level masking effect while the two models for gate-level stuck-at-0 AGEN faults over-estimate the μ arch-level masking effect.

In terms of coverage, the P- and PD-models do reasonably well for gate-level ALU stuck-at-0 fault and AGEN 1-cycle delay fault with differences less than 5%. However, for the other fault models, the P- and PD-models have 9+% differences in coverage.

In spite of extensive analysis and modeling, the probabilistic models do not accurately capture the μ arch-level behavior of gate-level faults due to the following reasons.

- The models are oblivious to temporal variation in the corruption rates, i.e., both the models use the probabilities of injecting k-bit flips as an average rate across all instructions for injections on a given wire.
- The probabilities on which the models pick the number of bits to flip, the pattern of the bit-flips, and the direction of the bit flips are not conditioned on the fault-free value on which the patterns are

applied. For example, although the pattern says that bit 1 should be flipped from a 1 to a 0, if the original value of the bit is 0, no flips occur. Thus, there are fewer flips than what the model expects, which skews the probabilities.

- The profiling runs consider the output value but overlook the input value that activates the fault in the circuit and produces the corrupted output.

As previously discussed, we derive a different model for each faulty run in SWAT-Sim that simulates a different fault in the gate-level circuit. However, for an abstract evaluation and accurate prediction, a unifying model that generalizes the proposed per-run models must be built. Based on the stated limitations of the P- and PD- models, an accurate unified μ arch-level model for the gate-level faults may be realizable. Nonetheless, until such a model is developed, SWAT-Sim remains an efficient platform for simulating and observing the system-level effects of gate-level faults.

From these results, we infer that μ arch-level stuck-at faults, do not, in some cases, accurately represent gate-level stuck-at or delay faults. Further, we have attempted to build probabilistic μ arch-level fault models based on the activation rate and corruption patterns of each run. However, the resulting models are inaccurate, in general, in representing gate-level faults. Since accurate microarchitecture-level fault models have yet been available, these findings show that simulating faults at gate-level is required to capture the accurate error effects of hardware faults. Because pure gate-level simulations often take very long time, techniques like SWAT-Sim are immensely useful to achieve both fault modeling fidelity and high speed that enables researchers to observe the software-level impact of hardware faults. Specifically, more accurate evaluations of current and future SWAT systems are achievable with the use of SWAT-Sim.

7.5 Summary and Discussion

As researchers realize the scaling-induced hardware reliability problem, many recent fault-tolerant solutions have been proposed from the microarchitecture level to the software level that offer high reliability at a low cost. To evaluate these schemes, many studies rely on statistical fault injection at the gate level or the microarchitecture level. However, fault simulation at the gate level, while sufficiently accurate, is notoriously slow. On the other hand, fault injection and simulation at the microarchitecture level is fast, but the accuracy

is questionable. In SWAT, because of the lack of efficient simulation tools for modeling the propagation of gate-level faults to software, we settled for conducting fault injection experiments at the microarchitecture level for evaluating the detection, diagnosis, and recovery modules of SWAT.

This chapter presents SWAT-Sim, a *fast* fault simulation infrastructure that *accurately* captures the impact of hardware gate-level faults on the executing software. SWAT-Sim uses the hierarchical simulation paradigm to achieve both speed and accuracy. By coupling a microarchitectural simulator with a gate-level simulator, SWAT-Sim does the slow but accurate gate-level simulation only when the faulty microarchitectural unit is utilized. This way, SWAT-Sim can achieve speeds close to microarchitectural simulation while modeling faults at the gate-level fidelity. The runtime invocation of gate-level simulation also allows SWAT-Sim to accurately capture the interaction between the software execution and the underlying persistent fault. To the best of our knowledge, SWAT-Sim is the first simulation framework that addresses this real-life behavior of faults. By employing a gate-level timing simulator, SWAT-Sim is also flexible to handle different kinds of timing fault models.

By implementing SWAT-Sim, we quantitatively compare the accuracy of microarchitecture-level fault models with gate-level fault models using the SWAT detection coverage and detection latency as proxies (since SWAT symptom detectors capture system-level effects of faults). For the microarchitectural structures we studied (Decoder, ALU, and AGEN), we found that microarchitecture-level fault models are inaccurate for several cases in our experiments. Through our detailed analysis, we attribute this inaccuracy to the differences in activation rate and bit corruption patterns. However, as these differences only impact some of cases reported in previous chapters, the qualitative results reported earlier in this thesis continue to hold.

With SWAT-Sim, we took a first step in deriving a potentially more accurate probabilistic microarchitecture-level fault models using data from SWAT-Sim. Our results, however, shows that these complex fault models are still inaccurate and unable to capture the complex manifestation of gate-level faults.

Overall, because accurate microarchitecture-level fault models have yet to exist, we believe that SWAT-Sim is an essential tool for capturing the software-level impact of hardware faults to help advance future work in SWAT and other work that needs observability of hardware faults at the software level. From the development of SWAT-Sim, we can see two future research directions. First, as much of current research has focused on investigating the in-field failure behavior of devices, these models can be distilled to the gate

level. Consequently, the fault models can be used inside SWAT-Sim to help processor and system designers understand the efficacy of new fault-tolerant solutions. Second, the issue of whether it is possible to model hardware failures at the microarchitecture level is a very interesting research question. In this work, although we did not derive an accurate microarchitecture-level fault model using data collected from SWAT-Sim, we did offer a number of ways to improve on the proposed models. In the end, we believe SWAT-Sim provides a very effective platform for powering hardware reliability research forward.

Chapter 8

Conclusions and Future Work

8.1 Conclusions

For a long time in the history of computing, CMOS has continued to scale according to Moore’s law, allowing ever-growing system integration and providing continuous performance improvement. On the other hand, as devices shrink perpetually, hardware failure rates are expected to increase due to a wide variety of error sources such as aging or wear-out, infant mortality induced by insufficient burn-in, transient errors caused by alpha particles from the packaging material and cosmic rays, design defects, and others. The pervasiveness of this growing reliability trend demands a low-cost in-field reliability solution that detects, diagnoses, recovers from, and/or repairs around failed components.

This thesis proposes a novel low-cost reliability solution that is based on the following two key observations. First, a hardware fault is only considered harmful if it affects software execution. Hence, an effective reliability solution only needs to handle hardware errors that propagate to the software. Second, even though the reliability threat is growing, fault-free operation still remains the common case. Therefore, reliability solutions must be optimized for fault-free operation.

Based on this design philosophy, this thesis presents *SWAT* (*SoftWare Anomaly Treatment*), a low-cost yet effective hardware reliability framework that minimizes the overall system cost by employing “always-on” error detection mechanisms that incur minimal cost in area, performance, and power. The tradeoff of the near-zero cost detection mechanism is to invoke a potentially high cost diagnosis mechanism to identify the error source after an infrequent event of an error detection.

In this thesis, we explore the design of the detection, diagnosis, and recovery mechanisms in *SWAT* and evaluate them using statistical fault injections. Our key experimental results are the following.

By employing simple, low-cost hardware-only monitors of software symptoms, the *SWAT* detectors are

able to achieve very high coverage for permanent faults, detecting 98% and 99% of the unmasked faults for SPEC workloads and server workloads, respectively, within 10 million instructions. For transient faults, while they are mostly masked, 59% of the unmasked faults for both workloads are detected, which is consistent with the previously proposed symptom-based detection schemes that handle only transient faults. Further, our detection mechanism also yields a very low SDC rate for both permanent and transient faults. These results clearly show that the SWAT detection approach is general and highly effective while incurring very low cost.

We also explore the use of likely program invariants for detecting permanent hardware faults and the resulting SDC rates improve dramatically by 73%. This shows software reliability techniques *can* be leveraged to ensure hardware reliability, strengthening the case for the SWAT approach.

For the detected faults, we invoke the trace based fault diagnosis algorithm to effectively identify the faulty microarchitectural structure. Our experimental results show that 98% of the faults are correctly diagnosed. By merely using one other core in the multicore system and the in-situ software execution, our diagnosis scheme is able to correctly diagnose most of the detected faults, including the faults in the datapath that were not addressed by prior work for in-field diagnosis.

For error recovery, to the best of our knowledge, this is the first work that quantitatively shows that both checkpointing and output buffering are equally important and necessary for full system recovery. With both schemes present, 99% of the detections for permanent faults injected into the hardware system running I/O intensive server workloads can be fully recovered. In addition, we also evaluate the tradeoffs made among the detection latency, the checkpoint interval of the checkpointing scheme, and the buffering interval of the output buffering method for optimizing the SWAT recovery strategy. Much prior work assumes that hardware checkpointing schemes would work at a reasonably low cost; our work on SWAT started with the same assumption but revealed through a closer analysis that the overhead involved could impact the overall system cost significantly. Our quantification prompts new techniques that find a better sweet spot between the output buffer size and the checkpointing overhead, possibly driven by lower detection latencies in SWAT. This is a subject of ongoing work.

The final contribution of this dissertation investigates the accuracy of microarchitecture-level fault modeling. We present a novel fast and accurate fault simulation framework, called SWAT-Sim, that is able to

capture the effect of gate-level hardware faults on the executing software. From our experiments, we found several cases where the microarchitecture-level faults are inaccurate in representing gate-level faults injected within the microarchitectural unit. Nevertheless, despite these differences, our qualitative findings on the detection, diagnosis, and recovery modules remain valid. With SWAT-Sim, future research in SWAT and other work in reliability will be able to accurately and efficiently capture the software-level effects of the underlying gate-level faults.

From the results on detection and diagnosis, we can clearly see that the SWAT approach is highly effective for ensuring hardware reliability while incurring very low cost. For recovery, we leveraged existing techniques used in other systems. A closer quantitative analysis revealed that while the overheads of these mechanisms are implementable, they are higher than for the rest of SWAT. Hence, there is still work to be done in this area. While efforts for developing and implementing the SWAT system continue, this thesis takes the first step to show how this low cost reliability solution can be realized. In addition, this dissertation also takes the first step in leveraging a well-known software bug detection technique for hardware reliability. This paves the way for deriving a truly error resilient system where hardware faults and software faults are treated the same in a unified framework. Eventually, we believe that the SWAT approach can make reliability affordable to the masses, ultimately neutralizing the impending reliability problem.

8.2 Limitations and Future Work

While the SWAT system is shown to be very effective to serve as the reliability solution for mainstream computing, there are many directions that can be explored in future work. The following discusses a few of the future research directions of SWAT.

8.2.1 Fundamentals of Symptom-Based Detection and Application-Aware Metrics for Evaluation

While the empirical results presented in this dissertation for the SWAT system are promising, there is still work to be done to understand the fundamental reasons for hardware faults to be detectable by software-level symptoms. By acquiring this knowledge, we can potentially characterize the behavior of hardware faults at the software level. This investigation will likely bring two very important prospects to SWAT error

resilient system design. First, by knowing how hardware faults behave at the software level, more potent symptom detectors can be derived. These new detectors would benefit the current SWAT system by (1) detecting the previously undetected faults and (2) reducing the detection latency for the currently detected faults. Second, this characterization can also identify the classes of hardware faults that are hard to detect. To handle these types of faults, SWAT can rely on other low-cost techniques (e.g., checkers embedded in the microarchitecture [45]) to ensure system reliability.

One way to acquire this knowledge is to investigate when the application state corruptions result in detections. In particular, for detection latency, this thesis uses the latency between the first architectural state corruption and the detection. However, this metric can be overly conservative because the corrupted architectural state could be masked by the software, as shown in our results.

Another way to characterize the effect of hardware faults on the software is to investigate cases that result in silent data corruptions. In this thesis, we conservatively classify a case to be an SDC if the resulting application output is different from the fault-free output. That is, if a single bit of the output is different, it is considered an SDC. However, some applications can tolerate outputs that have minor errors in them, as shown in [38]. Future work, therefore, can exploit the idea of soft computing for future generations of SWAT.

8.2.2 Implementation of SWAT

This dissertation shows the efficacy of the SWAT system in protecting future hardware in the simulation environment. Although we consider the simulated experiments sufficiently detailed, implementing an actual prototype of SWAT would only strengthen the case of designing resilient systems with the SWAT approach. As this process involves sorting out various system engineering issues in the real software and hardware environment, building such a prototyped SWAT system will not be an easy task. Nevertheless, showing a real system that detects, diagnoses, and recovers from hardware faults would certainly clear much skepticism about SWAT.

On the software side, developing the SWAT firmware is very important as it is responsible for coordinating the different fault-handling operations. For example, post detection, the SWAT firmware is woken up to diagnose the source of the error and to run the TBFD algorithm if necessary. In this process, the SWAT

firmware must be invoked at a non-symptom causing core because it can be corrupted by the possible persistent faults at the symptom causing core. In addition to this issue, SWAT system designers need to sort out the memory address space used by the firmware, the interface provided by the hardware to the firmware for diagnosing and recovering faults, the approach for ensuring reliable firmware execution, and so on. Further, to reap the most benefits of symptom-based detection, as shown in our invariant detection scheme, the applications may need to be modified for SWAT to achieve higher fault coverage. We believe that cultivating information from and enhancing both the applications and the operating system will greatly improve our SWAT implementation as well.

While the SWAT approach emphasizes very low hardware overhead, the hardware platform of SWAT still contains some hardware support. The following describes some of the hardware support needed in SWAT. In the detection module, symptom monitors such as the hang detector relies on a small hardware table to keep track of potential infinite loops. After a detection occurs, a special hardware interrupt needs to be delivered to another core to facilitate diagnosis. The TBFD algorithm itself requires a hardware buffer for the ITB to collect resource usage information from the faulty core efficiently. For error recovery, the ReVive checkpointing mechanism needs shadow architectural registers and enhancement to the cache subsystem. The output buffering mechanism requires certain amount of buffer storage to hold potentially faulty events. As part of the ongoing work towards building the SWAT error resilient system, the SWAT research group is currently developing an FPGA prototype of SWAT that addresses the various hardware needs described above.

8.2.3 Hardware Fault Models

As indicated by our work on SWAT-Sim, the microarchitecture-level fault models are in general inaccurate when compared to gate-level models. Hence, SWAT-Sim provides an efficient platform for fast and accurate hardware fault simulations. While the work presented in this thesis only looks at relatively simple stuck-at and delay fault models, we do not claim that these models capture the real-world behavior of in-field hardware faults. As much current research is focusing on different in-field failures caused by radiation, wear-out, process variation, and others, one important future direction is to incorporate these failure models into SWAT-Sim so as to accurately evaluate SWAT. Given the hierarchical nature of SWAT-Sim, future work

can extend SWAT-Sim to model failures at the circuit or device level (e.g., using SPICE simulations). This way, the effects of various hardware failure can be modeled realistically for the most accurate evaluation of SWAT and other fault-tolerant solutions.

Another direction we have explored in the SWAT-Sim work is the possibility of deriving accurate microarchitecture-level fault models. The motivation for deriving one such model is to allow accurate evaluations of fault-tolerant solutions without modeling hardware faults at gate-level and suffering the performance overhead. As there is an increasing number of hardware reliability solutions proposed at the microarchitecture level or higher, future investigations of efficiently modeling faults at the microarchitecture level will provide a convenient platform to enable accurate evaluations.

In this dissertation, we have focused primarily on in-core hardware faults. As the current error rate for this part of the mainstream processor core is not as high as caches or memory, it is relatively less protected. Going forward, the processor core is also relatively vulnerable to errors. Hence, our study concentrates on the processor core.

On the other hand, with an increasing number of cores integrated on chip, the off-core components (e.g., caches, on-chip interconnect, memory controller, etc.) make up a significant part of the processor. Hence, we believe investigating the efficacy of SWAT in terms of detection, diagnosis, and recovery on these off-core faults is an important piece of future work.

8.2.4 Multithreaded Workloads on Multicore Systems

Since this is the first work to show the complete workings of the SWAT system, we assume single-threaded applications running on a multicore system. With the widespread use of multicore systems, many applications are moving towards multithreaded designs to take advantage of the additional cores. In multithreaded workloads, as SWAT detects hardware faults when they propagate and manifest as software symptoms, it is unclear how fault propagation in the multithreaded execution would impact the processes of detection and diagnosis (note that the recovery mechanism targets multiprocessor systems and can naturally recover errors in this environment).

There are a few foreseeable issues that potentially create challenges for SWAT. First, while simple symptoms are shown to work well for single-threaded execution, it is unclear whether faults would manifest and

get detected in the same manner in the multithreaded environment. Second, because different threads of the multithreaded software may interact with each others, a fault can corrupt the thread that runs on the faulty core and propagate to another thread running on a fault-free core, causing a symptom on that fault-free core. Hence, the SWAT diagnosis process may not rely on the assumption that the symptom-causing core is potentially faulty. Third, as different instances of replays of the multithreaded execution may have different patterns of thread interleaving, faults may propagate to different cores in multiple replays from the same checkpoint. This creates additional difficulty in the diagnosis.

Given these challenges, future generations of SWAT must contain efficient and effective solutions for ensuring the reliability of multicore systems running multithreaded workloads in order to be broadly deployable. Recently, my colleagues and I investigated how to deploy SWAT on multicore systems running multithreaded workloads (not reported in this thesis) [26]. While we found that the SWAT symptom-based detection mechanism remained effective, we also observed that hardware faults did propagate across cores and got detected on a fault-free core. To ensure hardware faults can be correctly identified in spite of non-determinism introduced by multithreaded executions and cross-core fault propagation, we derived a novel fault diagnosis algorithm based on isolated (against propagation) deterministic replay (against non-determinism). Our results show that this new multicore diagnosis algorithm is able to correctly identify the faulty core in almost all cases. While this work takes the initial step to look into SWAT in multithreaded multicore environment, we believe future work can further improve the efficacy of both the detection and diagnosis mechanisms.

8.2.5 Recovery Mechanisms

For error recovery, this thesis quantitatively investigates the importance of both checkpointing and output buffering mechanisms. From our analysis on the overheads incurred by these methods, although the overhead of current solutions is manageable, it is relatively high when compared to the rest of the SWAT system. Specifically, the current detection latency of SWAT's symptom monitors demands a long recovery latency. The resulting storage overhead of the buffering mechanism would then be significant. As future generations of SWAT continue to improve in detection latencies, the buffering overhead can be reduced. On the other hand, the recovery latency can only be shortened by the use of shorter checkpoint intervals. However, the

performance overhead of ReVive for frequent checkpointing would then be high. While we consider the current techniques to be largely limited, this finding also presents a great opportunity for finding low-cost efficient checkpointing and buffering schemes in future work.

As we optimistically project that the detection latency of SWAT detectors continues to decrease, checkpoint recovery techniques from other fields of architecture research can potentially be leveraged. For example, there is a rich body of literature that studies different forms of speculative multithreading and transactional memory systems. Mechanisms proposed in these areas usually contain a form of efficient checkpointing mechanism that can roll back speculative executions that contain a small number of instructions. Hence, one potential checkpointing solution for future SWAT systems would be to rely on these solutions, which also amortizes the system cost.

While Chapter 6 has taken a close look at the need and the overhead of the output buffering mechanism, the actual buffering module has yet been modeled in detail. At the high level, the overhead of this module is closely related to the detection latency. If the latency is short, there will be fewer events and the buffer can potentially reside on-chip. Further, the buffering mechanism would potentially impact the throughput of the system. For future work, we believe it is essential to investigate the design of this module so that it can have minimal impact on area, power, and performance.

In this thesis, we have mainly focused on hardware-based techniques for recovery. Nevertheless, not unlike other fault-tolerant systems, there are always a small number of faults that are difficult and take very long to detect in SWAT. For these types of faults, it may be more advantageous to rely on system level or even application level checkpointing. As the software-based checkpointing schemes usually take coarse-grained checkpoints (e.g., once every hour), these faults may still be recoverable. Further, the executing software usually has better control over and knowledge of the I/O events than hardware. For some applications, the system can potentially be fully recovered even from these hard-to-detect faults. We leave the investigation of the hardware-software hybrid recovery mechanism as future work.

In summary, this thesis has investigated the SWAT approach for designing future error resilient systems, which is to ensure reliability through treating the software anomalies caused by the underlying faults. While we have found the SWAT system can offer high resiliency at a very low cost, there is still much work to be done. Through rigorous research on the areas stated above, having reliable computer systems for all

consumers at very low cost may become a reality in the near future.

References

- [1] *Intl. Technology Roadmap for Semiconductors*, 2008. update.
- [2] R.E. Ahmed, R.C. Frazier, and P.N. Marinos. Cache-aided rollback error recovery algorithms for shared memory multiprocessor systems. *Fault-Tolerant Computing, International Symposium on*, pages 82–88, 1990.
- [3] Todd Austin et al. Opportunities and Challenges for Better than Worst-Case Design. In *ASP-DAC '05: Proceedings of the 2005 conference on Asia South Pacific design automation*, pages 2–7, New York, NY, USA, 2005. ACM.
- [4] Todd M. Austin. DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design. In *International Symposium on Microarchitecture*, 1998.
- [5] David H. Bailey et al. Berkeley Lab Checkpoint/Restart. Website, 2008. <http://ftg.lbl.gov/CheckpointRestart/CheckpointRestart.shtml>.
- [6] David Bernick et al. NonStop Advanced Architecture. In *the Proc. of International Conference on Dependable Systems and Networks*, 2005.
- [7] Jason Blome et al. Self-calibrating Online Wearout Detection. In *International Symposium on Microarchitecture*, 2007.
- [8] Shekhar Borkar. Designing Reliable Systems from Unreliable Components: The Challenges of Transistor Variability and Degradation. *IEEE Micro*, 25(6), 2005.
- [9] Shekhar Borkar. Microarchitecture and Design Challenges for Gigascale Integration. In *International Symposium on Microarchitecture*, 2005. Keynote Address.
- [10] Fred Bower et al. A Mechanism for Online Diagnosis of Hard Faults in Microprocessors. In *International Symposium on Microarchitecture*, 2005.
- [11] Fred A. Bower, Daniel Sorin, and Sule Ozev. Online Diagnosis of Hard Faults in Microprocessors. *ACM Transactions on Architecture and Code Optimization*, 4(2), 2007.
- [12] Hungse Cha et al. A Gate-Level Simulation Environment for Alpha-Particle-Induced Transient Faults. *IEEE Transactions on Computers*, 45(11), 1996.
- [13] Kypros Constantinides et al. Software-Based On-Line Detection of Hardware Defects: Mechanisms, Architecture Support, and Evaluation. In *International Symposium on Microarchitecture*, 2007.

- [14] cURL. Tool for transferring files with URL syntax. Website. <http://curl.haxx.se>.
- [15] Edward W. Czeck and Daniel P. Siewiorek. Effects of Transient Gate-Level Faults on Program Behavior. In *International Symposium on Fault-Tolerant Computing*, 1990.
- [16] C. Dawson, S. Pattanam, and D. Roberts. The Verilog Procedural Interface for the Verilog Hardware Description Language. In *Verilog HDL Conference*, 1996.
- [17] Martin Dimitrov and Huiyang Zhou. Unified Architectural Support for Soft-Error Protection or Software Bug Detection. In *International Conference on Parallel Architectures and Compilation Techniques*, 2007.
- [18] George Dunlap et al. ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. In *International Symposium on Operating Systems Design and Implementation*, 2002.
- [19] Elmootazbellah N. Elnozahy and Willy Zwaenepoel. Manetho: Transparent roll back-recovery with low overhead, limited rollback, and fast output commit. *IEEE Trans. Comput.*, 41(5):526–531, 1992.
- [20] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Trans. Software Eng.*, 2001.
- [21] Michael D. Ernst et al. The Daikon System for Dynamic Detection of Likely Invariants. *Science of Computer Programming*, 2007.
- [22] O. Goloubeva et al. Soft-Error Detection Using Control Flow Assertions. In *Proc. of 18th IEEE Intl. Symp. on Defect and Fault Tolerance in VLSI Systems*, 2003.
- [23] Mohamed Gomaa et al. Transient-Fault Recovery for Chip Multiprocessors. In *International Symposium on Computer Architecture*, 2003.
- [24] Sudheendra Hangal and Monica S. Lam. Tracking Down Software Bugs Using Automatic Anomaly Detection. In *International Conference on Software Engineering*, May 2002.
- [25] Sudheendra Hangal and Monica S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the International Conference on Software Engineering*, 2002.
- [26] Siva Kumar Sastry Hari, Man-Lap Li, Pradeep Ramachandran, Byn Choi, and Sarita V. Adve. Low-Cost Permanent Fault Detection and Diagnosis for Multicore Systems. In *the Proc. of International Symposium on High Performance Computer Architecture*, 2009. To appear.
- [27] Allan Hirt. *Pro SQL Server 2005 High Availability*. Apress, Inc., New York, NY, USA, 2007.
- [28] Intel. World’s First 2-Billion Transistor Microprocessor. Website, 2008. <http://www.intel.com/technology/architecture-silicon/2billion.htm>.
- [29] Zbigniew Kalbarczyk et al. Hierarchical Simulation Approach to Accurate Fault Modeling for System Dependability Evaluation. *IEEE Transactions on Software Engineering*, 25(5), 1999.
- [30] G. Kanawati et al. FERRARI: A Flexible Software-based Fault and Error Injection System. *IEEE Computer*, 44(2), 1995.

- [31] Yoshio Kataoka, Michael D. Ernst, William G. Griswold, and David Notkin. Automated support for program refactoring using invariants. In *IEEE Int'l Conf. on Software Maintenance (ICSM)*, 2001.
- [32] Hue-Sung Kim, Arun K. Somani, and Akhilesh Tyagi. A Reconfigurable Multi-function Computing Cache Architecture. In *International Symposium on Field Programmable Gate Arrays*, 2000.
- [33] K. Li, J. F. Naughton, and J. S. Plank. Low-latency, concurrent checkpointing for parallel programs. *IEEE Trans. Parallel Distrib. Syst.*, 5(8):874–879, 1994.
- [34] Man-Lap Li, Pradeep Ramachandran, Ulya R. Karpuzcu, Siva Kumar Sastry Hari, and Sarita V. Adve. Accurate Microarchitecture-Level Fault Modeling for Studying Hardware Faults. In *the Proc. of International Symposium on High Performance Computer Architecture*, 2009.
- [35] Man-Lap Li, Pradeep Ramachandran, Swarup Kumar Sahoo, Sarita V. Adve, Vikram Adve, and Yuanyuan Zhou. Trace-Based Microarchitecture-Level Diagnosis of Permanent Hardware Faults. In *the Proc. of International Conference on Dependable Systems and Networks*, 2008.
- [36] Man-Lap Li, Pradeep Ramachandran, Swarup Kumar Sahoo, Sarita V. Adve, Vikram Adve, and Yuanyuan Zhou. Understanding the Propagation of Hard Errors to Software and Implications for Resilient Systems Design. In *the Proc. of International Conference on Architectural Support for Programming Languages and Operating Systems*, 2008.
- [37] X. Li, S. V. Adve, P. Bose, and J. A. Rivers. SoftArch: An Architecture-Level Tool for Modeling and Analyzing Soft Errors. In *the Proc. of International Conference on Dependable Systems and Networks*, June 2005.
- [38] Xuanhua Li and Donald Yeung. Application-level correctness and its impact on fault tolerance. In *HPCA '07: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 181–192, Washington, DC, USA, 2007. IEEE Computer Society.
- [39] Ben Liblit, Alexander Aiken, Alice X. Zheng, and Michael I. Jordan. Bug isolation via remote program sampling. In *Proc. of Conf. on Programming Language Design and Implementation*, 2003.
- [40] Ben Liblit, Mayur Naik, Alice X. Zheng, Alexander Aiken, and Michael I. Jordan. Scalable statistical bug isolation. In *Proc. of Conf. on Programming Language Design and Implementation*, 2005.
- [41] Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. AVIO: Detecting Atomicity Violations via Access Interleaving Invariants. In *the Proc. of International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.
- [42] Milo Martin et al. Multifacet's General Execution-Driven Multiprocessor Simulator (GEMS) Toolset. *Computer Architecture Newsletters*, 33(4), 2005.
- [43] Yoshio Masubuchi, Satoshi Hoshina, Tomofumi Shimada, Hideaki Hirayama, and Nobuhiro Kato. Fault recovery mechanism for multiprocessor servers. *Fault-Tolerant Computing, International Symposium on*, 0:184, 1997.
- [44] Carl Mauer et al. Full-System Timing-First Simulation. *SIGMETRICS Perf. Eval. Rev.*, 30(1), 2002.
- [45] Albert Meixner et al. Argus: Low-Cost, Comprehensive Error Detection in Simple Cores. In *International Symposium on Microarchitecture*, 2007.

- [46] Ali Mili. Towards a theory of forward error recovery. *IEEE Trans. Softw. Eng.*, 11(8):735–748, 1985.
- [47] Shahrzad Mirkhani, Meisam Lavasani, and Zainalabedin Navabi. Hierarchical fault simulation using behavioral and gate level hardware models. In *11th Asian Test Symposium*, 2002.
- [48] M. Mueller et al. RAS Strategy for IBM S/390 G5 and G6. *IBM Journal on R&D*, 43(5/6), Sept/Nov 1999.
- [49] Shubhendu S. Mukherjee, Christopher Weaver, Joel Emer, Steven K. Reinhardt, and Todd Austin. A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor. In *International Symposium on Microarchitecture*, 2003.
- [50] Jun Nakano et al. ReViveI/O: Efficient Handling of I/O in Highly-Available Rollback-Recovery Servers. In *the Proc. of International Symposium on High Performance Computer Architecture*, 2006.
- [51] Nithin Nakka et al. An Architectural Framework for Detecting Process Hangs/Crashes. In *European Dependable Computing Conference (EDCC)*, 2005.
- [52] Satish Narayanasamy, Gilles Pokam, and Brad Calder. BugNet: Continuously Recording Program Execution for Deterministic Replay Debugging. In *International Symposium on Computer Architecture*, 2005.
- [53] Jeremy W. Nimmer and Michael D. Ernst. Automatic generation of program specifications. In *Proc. ACM SIGSOFT Int’l Symp. on Software Testing and Analysis*, 2002.
- [54] Karthik Pattabiraman et al. Dynamic Derivation of Application-Specific Error Detectors and their Implementation in Hardware. In *European Dependable Computing Conference*, 2006.
- [55] Bipul C. Paul et al. Temporal Performance Degradation Under NBTI: Estimation and Design for Improved Reliability of Nanoscale Circuits. In *DATE*, 2006.
- [56] Andrea Pellegrini et al. CrashTest: A Fast High-Fidelity FPGA-Based Resiliency Analysis Framework. In *International Conference on Computer Design*, 2008.
- [57] Marius Pirvu, Laxmi Bhuyan, and Rabi Mahapatra. Hierarchical simulation of a multiprocessor architecture. 2000.
- [58] J. S. Plank, K. Li, and M. A. Puening. Diskless checkpointing. *IEEE Transactions on Parallel and Distributed Systems*, 9(10):972–986, October 1998.
- [59] Milos Prvulovic et al. ReVive: Cost-Effective Architectural Support for Rollback Recovery in Shared-Memory Multiprocessors. In *International Symposium on Computer Architecture*, 2002.
- [60] Feng Qin, Joseph Tucek, Jagadeesan Sundaresan, and Yuanyuan Zhou. Rx: treating bugs as allergies—a safe method to survive software failures. *SIGOPS Oper. Syst. Rev.*, 39(5):235–248, 2005.
- [61] Paul Racunas et al. Perturbation-based Fault Screening. In *the Proc. of International Symposium on High Performance Computer Architecture*, 2007.
- [62] R Raghuraman. Simulation Requirements For Vectors In Ate Formats. *ITC*, 00:1100–1107, 2004.

- [63] Pradeep Ramachandran et al. Statistical Fault Injection. In *the Proc. of International Conference on Dependable Systems and Networks*, 2008.
- [64] Pradeep Ramachandran, Siva Kumar Sastry Hari, Man-Lap Li, Sarita Adve, and Shobha Vasudevan. Reducing Silent Data Corruptions and Recovery Overheads with Software-Level Detectors. Submitted for publication.
- [65] V. Reddy et al. Assertion-Based Microarchitecture Design for Improved Fault Tolerance. In *International Conference on Computer Design*, 2006.
- [66] Steven K. Reinhardt and Shubhendu S. Mukherjee. Transient Fault Detection via Simultaneous Multithreading. In *International Symposium on Computer Architecture*, 2000.
- [67] George A. Reis et al. Software-Controlled Fault Tolerance. *ACM Transactions on Architectural Code Optimization*, 2(4), 2005.
- [68] Eric Rotenberg. AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors. In *International Symposium on Fault-Tolerant Computing*, 1999.
- [69] Giacinto P. Saggese et al. An Experimental Study of Soft Errors in Microprocessors. *IEEE Micro*, 25(6), 2005.
- [70] Swarup Kumar Sahoo, Man-Lap Li, Pradeep Ramachandran, Sarita V. Adve, Vikram Adve, and Yuanyuan Zhou. Using Likely Program Invariants to Detect Hardware Errors. In *the Proc. of International Conference on Dependable Systems and Networks*, 2008.
- [71] S. Sarangi et al. A Model for Timing Errors in Procs with Parameter Variation. In *Symp. on Quality Electronic Design*, 2007.
- [72] Design Panel, SELSE II - Reverie, 2006. <http://www.selse.org/selse2.org/recap.pdf>.
- [73] Smitha Shyam et al. Ultra Low-Cost Defect Protection for Microprocessor Pipelines. In *the Proc. of International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.
- [74] Daniel Sorin et al. SafetyNet: Improving the Availability of Shared Memory Multiprocessors with Global Checkpoint/Recovery. In *International Symposium on Computer Architecture*, 2002.
- [75] Lisa Spainhower et al. IBM S/390 Parallel Enterprise Server G5 Fault Tolerance: A Historical Perspective. In *IBM Journal of R&D*, September/November 1999.
- [76] Sudarshan M. Srinivasan, Srikanth Kandula, Christopher R. Andrews, and Yuanyuan Zhou. Flashback: A Lightweight Extension for Rollback and Deterministic Replay for Software Debugging. In *USENIX Annual Technical Conference, General Track*, pages 29–44, 2004.
- [77] Sun. OpenSPARC T1 Processor. Website, 2007. <http://www.opensparc.net/>.
- [78] Ramtilak Vemu and Jacob A. Abraham. CEDA: Control-flow Error Detection through Assertions. In *Intl. On-Line Test Symposium*, 2006.

- [79] Rajesh Venkatasubramanian et al. Low-Cost On-Line Fault Detection Using Control Flow Assertions. In *International Online Test Symposium*, 2003.
- [80] Virtutech. Simics Full System Simulator. Website, 2006. <http://www.simics.net>.
- [81] N. Wang et al. ReStore: Symptom-Based Soft-Error Detection. *IEEE Trans. on Dependable and Secure Computing*, 3(3), 2006.
- [82] Nicholas Wang et al. Characterizing the Effects of Transient Faults on a High-Performance Processor Pipeline. In *the Proc. of International Conference on Dependable Systems and Networks*, 2004.
- [83] Min Xu, Rastislav Bodik, and Mark Hill. A “flight data recorder” for enabling full-system multiprocessor deterministic replay. In *International Symposium on Computer Architecture*, 2003.
- [84] David Yen. Chip Multithreading Processors Enable Reliable High Throughput Computing. In *International Reliability Physics Symposium*, 2005. Keynote Address.
- [85] P. Zhou, W. Liu, F. Long, S. Lu, F. Qin, Y. Zhou, S. Midkiff, and J. Torrellas. Accmon: Automatically detecting memory-related bugs via program counter-based invariants. In *Proc. ACM/IEEE Int’l Symposium on Microarchitecture*, 2004.
- [86] Pin Zhou, Wei Liu, Fei Long, Shan Lu, Feng Qin, Yuanyuan Zhou, Sam Midkiff, and Josep Torrellas. AccMon: Automatically Detecting Memory-Related Bugs via Program Counter-based Invariants. In *International Symposium on Microarchitecture*, 2004.
- [87] Pin Zhou, Feng Qin, Wei Liu, Yuanyuan Zhou, and Josep Torrellas. iWatcher: Simple, General Architectural Support for Software Debugging. IEEE Micro Special Issue: Micro’s Top Picks from Computer Architecture Conferences, 2004.
- [88] Pin Zhou, Radu Teodorescu, and Yuanyuan Zhou. HARD: Hardware-Assisted Lockset-based Race Detection. In *the Proc. of International Symposium on High Performance Computer Architecture*, 2007.

Author's Biography

Man-Lap Li was born in Hong Kong. He received his Bachelor of Science degree in Electrical Engineering and Computer Science from the University of California, Berkeley in 2001, and the Master of Science degree in Electrical Engineering from the University of Illinois at Urbana-Champaign in 2005. He received the 2008 W. J. Poppelbaum Memorial Award from the Department of Computer Science at the University of Illinois at Urbana-Champaign in recognition of his research in computer architecture.