DETECTING AND RECOVERING FROM IN-CORE HARDWARE FAULTS THROUGH SOFTWARE ANOMALY TREATMENT

BY

PRADEEP RAMACHANDRAN

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, 2011

Urbana, Illinois

Doctoral Committee:

Professor Sarita V. Adve, Chair, Director of Research
Associate Professor Vikram S. Adve
Assistant Professor Samuel T. King
Professor Marc Snir
Pradip Bose, Ph.D., IBM Research

Abstract

Aggressive scaling of CMOS transistors has enabled extensive system integration and building faster and more efficient systems. On the flip side, this has resulted in an increasing number of devices that fail in shipped components in-the-field for a variety of reasons including soft errors, wear-out failures, and infant mortality. The pervasiveness of the problem across a broad market demands low cost and generic reliability solutions, precluding traditional solutions that employed excessive redundancy or piecemeal solutions that address only a few failure modes.

This dissertation presents SWAT (SoftWare Anomaly Treatment), a low cost resiliency solution that effectively handles hardware faults while incurring low cost during the common mode of fault-free operations. SWAT is based on two key observations about the design of resilient systems. First, only those hardware faults that affect software need to be handled and second, since the common mode of operation is fault-free, fault-free execution should incur near-zero overheads. SWAT thus uses novel zero to low cost hardware and software monitors that watch for anomalous software behavior to detect hardware faults. SWAT then relies on hardware support for checkpointing and rollback recovery. When dealing with fault recovery in the presence of I/O, we identify that existing software-level mechanisms that handle output buffering fall short. This dissertation therefore proposes a simple low-cost hardware buffer for output buffering and demonstrates that this strategy achieves high recoverability while incurring low overheads. Although not detailed in this dissertation, SWAT contains a comprehensive diagnosis procedure that is invoked in the rare event of a fault to isolate the root-cause of the fault by distinguishing between software bugs, transient hardware faults, and permanent hardware faults. Effectively, SWAT handles hardware faults uniformly as software bugs, amortizing the resiliency cost across both hardware and software reliability.

The results in this dissertation show that the SWAT strategy is effective to detect and recover the

system from a variety of in-core permanent and transient faults in various microarchitecture units for both compute-intensive and I/O-intensive workloads. In particular, this dissertation demonstrates that the SWAT detectors detect nearly all permanent and transient faults in most hardware units in both types of workloads, with only a small fraction of the faults corrupting application output. (Certain hardware structures like the FPU may need additional support to be amenable to software anomaly detection.) Further, a majority of these faults are tolerated by the applications due to their inherent fault-tolerant nature, resulting in only 0.2% of the injected faults affecting the application and yielding incorrect outputs (such faults are classified as Silent Data Corruptions, or SDCs). When attempting to recover the detected faults, we show that handling I/O is important for fault recovery. With our proposed low-cost hardware for output buffering, we show that over 94% of the detected faults are recoverable with low performance and area overheads during fault-free execution even in the presence of I/O. Finally, this dissertation builds a fundamental understanding behind why the SWAT strategy is effective for handling faults in modern workloads. The key insight is that the SWAT detectors are adept at detecting perturbations in control operations and memory addresses and a majority of the application values affect such operations. Faults in values that that never affect such operations are hard-to-detect and require additional support to be amenable to software anomaly detection.

In summary, this dissertation presents SWAT as a complete solution to detect and recover from from in-core hardware faults. The techniques presented here therefore have far reaching implications on the design of low-cost solutions to handle unreliable hardware.

For Suchi

Acknowledgments

I have waited for six long years to write this chapter of my thesis and thank everyone who has helped me achieve my most arduous goal to date, but now that I am here, I am lost in memories and my words are failing me. I imagine that I can never do justice to all the sacrifices that people have made for me in this small acknowledgments section, but I will, as I have always done in life, put my best step forward.

No language has enough words to capture the gratitude that I feel towards my adviser, Sarita Adve. Her calm and composed approach towards technically challenging problems and her never-say-die attitude to attain the optimal solution has groomed me to become an energetic and innovative researcher. Apart from her work (which she is absolutely brilliant at), I adore her many values towards life at large and hope that some day, I will be as successful a researcher and as nice a person as she is today. Thanks Sarita, its been a pleasure and an honor to work under your guidance.

I would also like to thank my lab-mates, Alex, Siva, Byn, Hyojin, Rakesh, Rob, Vibhore, and Jayanth for a healthy research environment and many collaborations. Special thanks to the members of my committee, Prof. Vikram Adve, for many discussions about the SWAT project and beyond, Prof. Sam King for valuable feedback on my work and the friendly discussions in the hallway about conference deadlines and the paper-beard, Prof. Marc Snir, for detailed comments on my thesis, and Dr. Pradip Bose for being a part of my research for the last six years and my manager when I was an intern at IBM.

Many thanks to the several support staff in the department including Molly (thanks for the occasional free coffee), David Anderson (you are a genius at bringing our cluster out of trouble), Andrea (for the never-ending list of conference room reservations that I request), and Mary Beth (for helping me with the deposit process) for their support. My research has been supported by

the NSF grant numbers CCF 0541383 and CCF 0811693, the Gigascale Systems Research Center (funded under FCRP, an SRC program), an Intel PhD Fellowship, and an IBM PhD Scholarship.

In the barren landscape of Champaign, I have had the honor of meeting the most wonderful set of people who forever will stay in my heart. This dissertation might have been ready a little sooner were it not for these folks, but the experience would not have been anything close to this magical. Each and every one of them has touched my heart every single day and has helped me keep my barely-there sanity through the years. Thanks to Dog, Harini, Poba, Kaushik, Pitu, Chandu, Aari, JP, Raghu, AJ, Vivek, Milu, Aloo, Bedhi, Vids, and Thathu. I could never have done it without all of you!

I would also like to acknowledge the support that I have received from my family from India – Amma, Appa, Praveen, Ammamma, Archana, Shraddha, Latha, Periyamma, Periyappa, and Dilip. They have helped me persevere through all these years to attain what seemed like an impossible mountain. Thanks for having the faith in my abilities and for teaching me how to dream.

Last, but certainly not the least, I would like to acknowledge the contribution of that one special person in my life – Suchi. My closest friend, my best bud, and the only person in the world who understands and appreciates my madness in all its vibrant colors. She has given me strength, courage, and reason to finish this dissertation. I hope that some day, I can do for you all the wonderful things that you have done for me and as a small token of my appreciation towards all that you have done for me, I dedicate this dissertation to you. Thanks!

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

The number of transistors on a given piece of Silicon has seen an exponential growth over the last few decades, as predicted by Moore's law [45]. This trend in technology scaling has lead to ever-increasing integration in the processor core, resulting in faster systems. This has fueled a computing revolution, resulting in the use of computers in all walks of life and affecting nearly every action in our daily lives.

This revolution, however, has a dark side. Although the scaled transistors are faster and cheaper, they are prone to failures due to their smaller dimensions. As we enter the late CMOS era where device dimensions approach atomic distances, components in shipped chips are expected to fail for a variety of reasons including soft-errors due to radiation from cosmic particles and packaging material, wear-out failures due to aging and elevated operating temperatures from increased power density, infant mortality due to insufficient burn-in, and design defects due to highly complex integrations [8, 79].

There is thus an impending challenge to the design of computer systems – one of building reliable systems from such unreliable hardware [8]. Effective fault tolerance mechanisms to detect such failures, diagnose their root cause, recover the system, and repair/reconfigure around the failed components are thus required. Further, this problem of unreliable hardware is expected to pervade the entire computing market, warranting solutions that incur low performance, area, and power costs to be deployable across a broad market.

High-end systems, such as main frames, and server systems, have employed methods that achieve high reliability and availability targets (of five 9's of reliability or more) for several decades now [7,

39, 72]. In order to achieve such high reliability targets, these systems deployed extensive amounts of redundancy in hardware and in software to detect and recover the system from faults. Dual modular redundancy (or its generalization, n-modular redundancy) is a commonly used mechanism for hardware redundancy where the hardware is duplicated (or has n copies in the n-modular system) to check for inconsistencies during execution. Software duplication through N-version programming has also been employed by such systems to ensure that the executing software does not contain any bugs [5]. Although these mechanisms achieve high reliability targets, they incur excessively high overheads. The dual-modular solution, for example, incurs an overhead of 100% in core area and power, and significant overheads in performance [4].

The challenge of unreliable hardware is, however, expected to affect consumer systems of the future. Such systems have stringent requirements in cost, making traditional solutions that employ excessive redundancy inviable due to their high overheads. Such systems may in fact be willing to accommodate slightly lower reliability targets at the benefit of lower performance, power, and area overheads. In a recent workshop, an industry panel converged on a 10% area overhead target to handle all sources of chip errors as a guideline for academic researchers [67].

1.2 SWAT – An Error Resilient System

This thesis presents SWAT (Software Anomaly Treatment) – a complete low-cost solution for in-core fault resiliency for commodity systems. SWAT encompasses strategies for detection, diagnosis, and recovery of permanent and transient hardware faults in both single and multicore systems [28, 33, 34, 35, 66].

Two high-level observations drive the SWAT work. First, a hardware reliability solution should handle only those faults that affect software execution. Second, despite the growing threat of reliability, fault-free operation remains the common case and must be optimized.

Motivated by these observations, SWAT detects faults by watching for anomalous software behavior with zero to low-cost hardware and software monitors. Fault detection is thus largely oblivious to the underlying mechanisms that causes the fault, treating hardware faults analogous to software bugs and potentially amortizing the overheads for system reliability. Once a fault is

detected, SWAT relies on checkpointing support for rollback recovery. Since committed external outputs cannot be rolled-back for recovery, externally visible outputs are buffered until they are verified to be fault-free (commonly referred to as the *output-commit problem*). Given that fault detection happens at a high level, diagnosis is required to identify the root cause of the fault by distinguishing between software bugs, transient hardware faults and permanent hardware faults [28, 34]. SWAT uses repeated rollbacks for both single-threaded and multi-threaded workloads to achieve this distinction. For permanent hardware faults, SWAT first diagnoses the faulty core in the multi-core environment [28], and then the faulty microarchitecture-level component within the faulty core [34]. SWAT thus relies on low-cost hardware and minimal support from software for fault detection and on a thin firmware layer to orchestrate diagnosis, recovery, and repair once a fault is detected.

Figure 1.1 gives a high level overview of SWAT. The SWAT system takes periodic checkpoints of the execution. When a *fault* in the underlying hardware is activated by the software and manifests as an *error*, the execution of the software is corrupted (shown in dotted lines). The SWAT detectors monitor for such anomalous execution of the software and detect the underlying hardware fault through a *Software Anomaly Detection*. If the fault does not cause any anomalous execution, the fault is successfully ignored.[1] Subsequently, the SWAT firmware is invoked to *diagnose* the fault, and to take the appropriate *recovery* action to ensure continuous system operation. For permanent faults, *repair* may also be invoked on the faulty component.

Compared to prior work on low-cost solutions that handle hardware faults, SWAT is a complete solution for fault resiliency that is not tailored to any particular fault model. Much of the prior work has focused only on detection and for faults from a particular fault model, such as transient faults [26, 55, 61, 63, 64, 75, 78], software bugs [21, 52] or permanent faults [12]. Further, most prior solutions tackle only one component (i.e., detection, diagnosis, or recovery) at a time, without considering its interactions with the other components. This may result in ignoring certain interactions that are critical to the design of the whole system (such as the relationship between fault detection and recovery), and may also result in high overheads when all the components are assem-

[1] A small fraction of the anomalous executions may escape SWAT's detectors. The detectors are designed such that these instances of escaped faults are minimized.

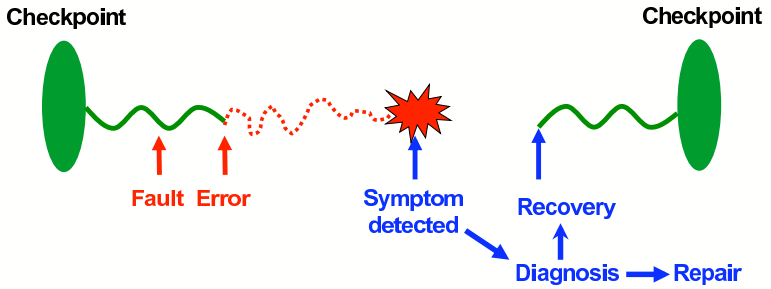


Figure 1.1: High-level overview of SWAT. SWAT uses software anomalies to detect faults that manifest as errors on software, and invokes diagnosis and recovery to ensure continuous system operation.

bled together for a full solution. Finally, SWAT is the first solution that proposes implementing output buffering with simple hardware. While much of the prior work has ignored output buffering, the only existing solution for buffering outputs in a modern system that relies on checkpointing for recovery is implemented in software [51] and suffers from fundamental limitations that make the committed outputs vulnerable to in-core faults.

1.3 Advantages of SWAT

Owing to the above strategy to handle failures, SWAT has the following advantages, overcoming several key limitations of existing solutions.

1. **Handles faults that matter.** Since SWAT uses software anomaly detectors that detect faults at a high level, it is oblivious to underlying failure modes. Further, the SWAT strategy results in ignoring faults that are masked at lower levels of the system such as the circuit, the microarchitecture, the architecture and the application. SWAT thus handles only those faults that matter, reducing overheads.

2. **Holistic systems view enables novel solutions.** The detection, diagnosis, and recovery components in SWAT are built in unison, with each component exploiting synergies from its interactions with other components. This holistic view enables novel solutions that were previously not feasible [66], providing avenues for further innovation.

3. **Low, amortized overheads.** The SWAT detectors are implemented with low-cost hardware monitors that do not affect the fault-free operations of the system. The philosophy of SWAT to optimize for the common case of fault-free operations thus results in a low-cost solution. Further, SWAT employs solutions (some of which are derived from prior work on handling software bugs) to handle a variety of hardware faults, amortizing the cost of resiliency across different hardware failure modes.

4. **Customizable and flexible.** The detection module of SWAT reports to a thin firmware layer that orchestrates the diagnosis, recovery, and repair processes. This firmware layer sits under the software and is customizable by the hardware vendor based on the needs of the system. (The hardware vendor thus does not have to entirely rely on the software vendor for resiliency, although the software may provide some hooks to observe anomalous software execution.) The firmware can be used to implement runtime trade-offs between overheads during fault-free execution and the achieved resiliency targets. This provides an additional degree of flexibility to the system, making the firmware-based strategy of SWAT attractive.

5. **Beyond hardware reliability.** While the focus of SWAT has been largely on hardware faults, the long term goal in SWAT is to build a unified solution that treats the problems of unreliable hardware and software as one. Further, some concepts in SWAT, especially in the diagnosis components [28, 34] may also have applications in post-Silicon test and debug cycles by providing debug traces that activate the underlying fault.

1.4 Contributions of This Thesis

The main contribution of this dissertation is presenting SWAT as a solution that detects and recovers from in-core hardware faults, along with an intuition behind its effectiveness. We do not evaluate SWAT on faults in the memory subsystem as techniques such as parity and ECC are commonly deployed to protect the system against such faults while the logic is typically not well-protected. For faults within the core, the SWAT detection strategy is demonstrated on various types of workloads, including compute-intensive, I/O-intensive, and media workloads [28, 35, 66]. The recovery module

orchestrates fault recovery in the presence of I/O, effectively handling the notorious *output commit problem*.

Other members of the SWAT team led the development of the SWAT diagnosis module that handles fault diagnosis in both single-core and multi-core systems has been developed [28, 34]. The diagnosis module leverages checkpointing support from the recovery module for repeated rollback-replays and first identifies whether the software anomaly detection was because of a software bug, a transient hardware fault or a permanent hardware fault. In the case of permanent hardware faults, it first identifies the faulty core in a multi-core environment without assuming a spare fault-free core [28], and then refines the diagnosis to a finer microarchitecture-level granularity using Trace-Based Fault Diagnosis (TBFD) [34]. Additionally, the diagnosis module may also be used to identify false positives from the heuristic software anomaly detectors. The diagnosis module is not detailed in this thesis, but interested readers can refer to the relevant paper for the single-threaded [34] and multi-threaded [28] diagnosis modules in SWAT.

1.4.1 Low-Cost Software Anomaly Detection

This thesis presents a collection of low-cost software anomaly detectors that are effective in detecting both permanent and transient hardware faults in compute-intensive SPEC and I/O-intensive server workloads. The detectors observe anomalous execution of the software with minimal support from the software (in the form of hooks that declare locations of key software-level events such as signaling mechanisms and abort routines) and are implemented with low-cost hardware (such as existing performance counters in modern processors).

Our results unequivocally show that this handful of simple detectors effectively detect most of the injected permanent and transient faults, while incurring low overheads during fault-free execution. A majority of the injected permanent and transient faults (injected at randomly chosen locations within the processor core and at random points during the execution of the application) that are not masked by these workloads are detected, and only a small fraction of the faults that escape detection corrupt the outputs of the application; such faults are called silent data corruptions (or SDCs). The server workloads have a resulting potential-SDC rate of under 0.2% and 0.4% for

permanent and transient faults, respectively, while the rates for the SPEC workloads are 0.6% and 0.6% for permanent and transient faults injected into non-FPU units, respectively. Further, over 98% of the faults detected in non-FPU units have a detection latency of under 10 million instructions.

1.4.2 Application-Aware Silent Data Corruptions

Since software anomaly detectors, like SWAT, detect hardware faults by observing anomalous software behavior, the properties of the application need to be considered to accurately gauge their limits. Taking such application-level fault tolerance properties into consideration, we show that most of the faults that escape fault detection and corrupt application outputs are tolerated by the application. For the server workloads, since network failures are common, the clients codes have fault tolerance built-in, in the form of repeated retries with back-off, that may be used to tolerate faults. For the SPEC codes, we follow previous work [17, 36] and show that many of these codes perform soft computations that can handle some degradation in output quality. Although the latter observation has been made previously, our analysis spans all 16 C/C++ workloads of the SPEC CPU 2000 suite and is thus more comprehensive. After considering application-level tolerance, we find that only 56 of the 35,760 faults injected into the server and SPEC workloads ($< 0.2\%$) are SDCs (assuming 1% quality degradation is acceptable for the soft computations in SPEC). This translates to a two orders of magnitude reduction in the FIT rate when compared to the base system with no protection against faults (all faults that are not masked lead to failures in the base system). Further, several of these 56 faults may be identified with additional support from the software, but we leave this exploration to future work.

1.4.3 Application-Aware Detection Latency

We next consider application-level metrics for the detection latency and understand its implications on fault recovery. We identify that although a fault may corrupt the architecture state, the system may still be recoverable as long as the affected state does not perturb software state. We thus redefine detection latency as the time from software state corruption (vs. architecture state corrup-

tion) to detection and show that although a shorter checkpoint intervals is sufficient, ignoring this distinction leads us to choosing checkpoint intervals of millions of instructions where the overheads from checkpointing are several orders of magnitude higher. It is thus important to consider an application-aware notion of detection latency to accurately evaluate the efficacy and the overheads of fault recovery.

1.4.4 Hardware Output Buffering for Fault Recovery

Previous work has largely ignored buffering external outputs to handle the *output commit problem* although it is important for any fault tolerance scheme that relies on checkpointing for fault recovery. We present and evaluate, for the first time, a hardware implementation of output buffering that circumvents fundamental limitations of previously proposed solutions that buffered outputs in software. Our results show that at short checkpoint intervals of up to 100K instructions, the proposed technique incurs <5% performance overhead on fault-free execution and incurs area overhead of under 2KB. At checkpoint intervals of millions of instructions, however, the incurred performance overheads are much higher, with up to a 62X increase in client response time for a checkpoint interval of 10M instructions. Practical systems should thus support short checkpoint intervals of under millions of instructions (milliseconds of execution in a modern processor) to minimize the fault-free overheads. Prior work on hardware checkpointing assumed, however, that intervals of 10s of milliseconds may be acceptable as they did not consider the overheads incurred from buffering outputs.

1.4.5 A Solution for Detecting and Recovering In-Core Hardware Faults

Putting the detection and the recovery modules together, we demonstrate a system that achieves low SDC rates and high recoverability for in-core hardware faults, while incurring low overheads during fault-free execution. To our knowledge, this is the first evaluation of a full recovery scheme with a software anomaly detection technique that detects and recovers from in-core faults even in the presence of I/O; prior work has evaluated solutions for either detection or for recovery in isolation. In this combined system, at a recovery interval of 100K instructions for server workloads, 94% of the

injected permanent and transient faults are either masked or recovered without affecting application output, 4% are detected but not recovered, and only 44 (0.2% of the injected 17,920) faults corrupt the application outputs and result in SDCs. We also demonstrate that output buffering and device recovery are critical for system recovery in the presence of I/O.

Given this high recoverability at short checkpoint intervals, SWAT can protect systems that see less than one fault every 10 milliseconds. Since the latency to fault detection and recovery is of the order of 100K instructions for a majority of the injected faults (which translates to sub-millisecond intervals in a modern 1GHz processor with an IPC of 1), and diagnosis takes up to 10 milliseconds to identify the root cause of the fault [28], SWAT can handle such high fault rates.

1.4.6 A Qualitative Understanding of Why SWAT Works

Finally, this dissertation builds an intuition behind why SWAT works by taking both an application-centric and a hardware-centric view on faults. With the application-centric view, it shows that $< 6\%$ of application values in modern applications do not affect control flow or memory addresses (we classify such faults are *data-only*). Consequently, a majority of the injected faults affect control operations or memory addresses and since software anomaly detectors are adept at identifying such perturbations, software anomaly detection is effective for fault detection. It also shows that faults in such data-only values are hard to detect and require additional support from techniques that are devised to identify faults in data values [66]. With a hardware-centric view, the dissertation also shows that faults in structures used for control operations are easier to detect than those that affect pure data low, and that a uniform round-robin schedule may help fault detection by making fault activation more uniform.

1.5 Organization

This dissertation is organized as follows. Chapter 2 presents work related to the concepts discussed in this thesis. Chapter 3 discusses the techniques for fault detection, along with an evaluation of their efficacy to detect permanent and transient hardware faults. Chapter 4 takes an application-aware notion towards fault tolerance and re-visits the SDC rate and the detection latency for the

SWAT detectors. Chapter 5 discusses fault recovery in SWAT, with particular emphasis on fault recovery in the presence of I/O. It also ties the recovery module along with the detection module and presents the efficacy of SWAT to detect and recover in-core hardware faults. Chapter 6 then builds a fundamental understanding behind why SWAT works by taking an application- and hardware-centric view on faults. Chapter 7 summarizes this dissertation and concludes with a discussion on directions for future work.

Chapter 2

Related Work

The SWAT system presented in this thesis is a low-cost solution encompassing detection, recovery, and diagnosis in the presence of permanent and transient hardware faults. Although there has been prior and concurrent work related to SWAT, to the best of our knowledge, SWAT is the first low-cost solution that provides a comprehensive solution for detection, diagnosis, and recovery of both permanent and transient hardware faults.

This chapter outlines related work with respect to fault detection (Section 2.1), application aware metrics (Section 2.2), fault recovery (Section 2.3), modeling application-level fault propagation (Section 2.4), and other related work (Section 2.5).

2.1 Fault Detection

2.1.1 Hardware-Only Detectors

Traditional Detectors

Traditional high-end commercial systems often detect faults through coarse-grain hardware-level redundancy (e.g., replicating an entire processor or a major portion of the pipeline) [7, 47, 72]. Such approaches incur significant area, performance, and power overheads as a significant portion of the hardware is duplicated.

Low-Cost Hardware Detectors

In an attempt to reduce the cost incurred from full hardware duplication, several approaches have been proposed to selectively place hardware-only detectors that detect hardware faults. Austin proposed DIVA, an efficient checker processor that is tightly coupled with the main processor's

pipeline to check every committed instruction for errors while incurring lower overheads than replicating the entire processor [4]. Argus proposed using computation checkers to detect hardware faults by monitoring their manifestations in data-flow, control-flow, and memory address values [44]. The IBM POWER6 system also uses hardware checker circuits to detect hardware faults by monitoring low-level hardware events [38]. Although these solutions lower the cost incurred from hardware added for resiliency, the overheads may still by significant. Argus, for example, reported a 16% area overhead for the core from the extra logic for fault detection [44]. Further, such low-level detection mechanisms may not identify faults in all hardware components (only those components that have checkers are protected) and may identify faults that are masked by higher levels of the system, such as the architecture or the application, incurring unnecessary overheads.

Shyam et al. recently proposed online testing of certain structures in the microprocessor for hard faults and recovery by disabling them and rolling back to a hardware checkpoint [68]. Since these tests are run only when the structures are idle, the performance loss incurred is rather small. Constantinides et al. enhanced this scheme further in by adding hardware support so that the software can control the online testing process, adding flexibility for choosing test vectors [12]. The performance penalty incurred by this form of software-controlled online testing is high for reasonable hardware checkpointing intervals (which are of the sub-millisecond range) as shown in Chapter 5. Furthermore, the continuous testing of hardware may accelerate the wear-out process, exacerbating the problem of failing hardware.

2.1.2 Symptom Detectors

Since the problem of unreliable hardware is expected to even affect commodity systems where some trade-offs in fault coverage for lower overheads may be acceptable, detectors that employ excessive redundancy may not be applicable owing to the high incurred cost.

This has resulted in a surge of low-cost solutions that monitor for software misbehavior or anomalies as symptoms of faults in the underlying hardware [12, 21, 26, 52, 55, 61, 63, 64, 75, 78]. Compared to traditional detectors, such detectors incur low cost in the common mode of fault-free operation, making them attractive alternatives. While these solutions have shown the efficacy of

symptom detectors to handle transient faults [26, 55, 61, 63, 64, 75, 78] and software bugs [21, 52], permanent hardware faults have been less studied [12]. Further, these solutions are largely piecemeal and are typically tailored for a particular type of fault, ignoring the effects of other faults in the system. The SWAT system, on the other hand, reduces the cumulative costs by not distinguishing between transient and permanent faults during fault detection and by devising a solution that encompasses all aspect of fault tolerance, including detection, recovery, and diagnosis.

2.2 Application-Aware Fault Tolerance Metrics

Previous work has observed that certain applications have an additional degree of fault tolerance by virtue of their computations [17, 36] and because they are recoverable even from corrupted architecture state, as long as the state of the application is pristine [1, 2, 10, 25].

We adapt previous work on application-level fault tolerance of multimedia [17, 36], artificial intelligence [36], and compute-intensive workloads [36] and derive acceptance criterion for the set of SPEC workloads and the server workloads that we use. We expand on prior work that observed that although commonly regarded as being intolerant to faults, even some of the SPEC workloads may tolerate corruptions in output values [36]. We demonstrate that only 4 of the 16 C/C++ codes in the SPEC suite do not tolerate corrupted outputs.

Further, prior work on application-level reliability has been largely restricted to using software assisted checkpointing techniques to record and replay the minimal state required for application-level recovery [1, 2, 10, 25, 32, 36, 69]. The focus of using application-aware notions to study the detection latency in this thesis is, however, to make the distinction between correct execution from an architecture and from an application stand point, and understand its impact on recoverability. While a similar observation was made by Li et al. [36], the focus of their work was on recovering the application by checkpointing minimal amount of state (their checkpoint constitutes registers state, PC, and stack). In this work, we are more concerned with identifying the checkpoint intervals for fault recovery; prior work on reducing the size of the checkpoints can still be leveraged.

2.3 Fault Recovery

Once a fault is detected, the system may be recovered through either *forward-error recovery (FER)*, or *backward-error recovery (BER)* techniques. In FER, the detectors provide sufficient information that enable fault recovery without re-execution. ECC is a good example of an FER scheme where single bit-flips are easily identified and corrected with ECC and without re-execution. BER schemes, on the other hand, rely on replaying the execution from a pre-recorded pristine checkpoint to restore the system. BER schemes therefore take periodic checkpoints of the system state, and buffer external outputs until they are verified to be fault-free to avoid the *output commit problem*.

SWAT relies on backwards-error recovery mechanisms for fault recovery, with support for checkpointing and output buffering. Since the detectors detect faults by observing software-level anomalies, the fault is allowed to corrupt the state of the system which is then recovered from a pristine checkpoint.

While BER techniques for fault recovery and support for transaction abort in transactional systems [46] bear some similarities, there are two key differences between the two solutions. First, owing to the limited abilities in I/O handling and limited buffering capabilities in caches, transactions are typically limited to be for short durations (of 10s or 100s of instructions). Fault-tolerant systems, on the other hand, warrant recovery for longer intervals, requiring explicit support for buffering outputs. Second, the frequency with which fault recovery is invoked for fault tolerant systems is much lower than that in transactional systems as faults are much rarer than transaction aborts. Transactional memory systems have nevertheless borrowed several ideas from the literature on BER techniques for fault recovery to enable low cost recovery when a transaction is aborted.

2.3.1 Hardware Versus Software Checkpointing

Software-level checkpointing schemes [11, 32, 41] are typically used for low frequency checkpointing (every few seconds or minutes) as they incur high overheads from operations for checkpointing. Consequently, they add unacceptable delays to I/O operations, significantly impacting fault-free execution. We therefore focus on hardware checkpointing schemes [16, 51, 60, 71] as they can handle high frequency checkpointing (every few milliseconds, or lower) at lower performance overheads.

2.3.2 Recovery without I/O

SafetyNet [71] and ReVive [60] are two schemes that handle recovery with hardware checkpointing without any application or ISA changes. Both schemes rely on a periodic snapshot of the architecture registers for processor checkpointing. SafetyNet collects its memory logs in dedicated hardware buffers called CLBs (Checkpoint Log Buffers), incurring hardware overheads. For checkpoint intervals of 1000s of instructions SafetyNet incurs negligible performance impact with a CLB sized at 512KB. ReVive, on the other hand, collects its memory logs in memory and protects them with parity. ReVive incurs much lower hardware and negligible performance overheads for checkpoint intervals in the 10s of milliseconds range. Recent proposals have also suggested using a combination of hardware and software for system recovery [16]. These schemes, however, do not handle recovery in the presence of I/O, making them unusable for any real-world application where I/O to crucial to communicate with the external world (either through inputs or through outputs).

2.3.3 Recovery with I/O Handling

In addition to architecture state recovery, output buffering and device recovery become important in the presence of I/O. Traditional fault-tolerant systems used replicated executions (with duplicate processors) to check the values and the order of I/O operations before committing them [7, 39]. To our knowledge, ReVive-I/O is the first and only implementation of output buffering in a modern checkpointing-based system without relying on such redundant executions [51].

For *output buffering*, ReVive I/O relies on a dedicated software layer called Pseudo Device Drivers (PDDs) [41] that buffers outputs in memory. These outputs are protected from faults by ReVive's memory checkpointing scheme. When the outputs are verified to be fault-free (i.e., at the next checkpoint interval), the PDD invokes the device driver to commit the outputs to the devices. There are, however, three limitations to such a software-level implementation. First, ReVive I/O assumes that the driver that actually commits the outputs to the devices does not activate the fault and that the committed outputs are fault-free. However, since the buffered outputs are high-level events (such as DMA writes to disk or to the network card), the driver may take tens of milliseconds to commit them to the devices [51]. In this interval, the driver software may activate an in-core

fault and commit corrupted outputs. This a fundamental, yet subtle, limitation of software-level buffering that has not been previously explored. Second, the PDD software that buffers outputs is complex as it requires understanding the output semantics of the devices. It is thus hard to extend this complex buffering to support generic devices. For example, ReVive I/O cannot handle outputs that by-pass kernel interfaces or use dedicated hardware. Finally, the lowest checkpoint interval that ReVive I/O has been evaluated for is 20ms for which outputs are delayed by 40ms. Since the acceptable response time for a latency-bound transaction is 100ms [53], the 40ms addition significantly reduces the amount of time available to handle the transaction, making the delays excessive. In fact, through the techniques presented in this dissertation, current symptom detectors can support shorter checkpoint intervals (of 100s of kilo-instructions) at which software-level solution like ReVive I/O have not been evaluated.

For *device recovery*, ReVive I/O resets the devices and reinitializes the drivers to a state corresponding to the reset device. In order to bring the reset devices (and drivers) to a state consistent with the current execution, the PDD then recommits outputs buffered (in software) from the previous recovery interval. Prior work on Shadow Drivers also relies on a similar strategy to recover the device and the driver from transient faults in the device [30, 74]. Both ReVive-I/O and the Shadow Drivers then rely on higher level protocols to replay inputs and handle duplicate outputs from recovery operations. Since the software for device recovery can be restored with the hardware checkpoint, our system relies on these schemes for device recovery.

2.4 Modeling Application-Level Fault Propagation

There is growing interest to model how hardware faults propagate through the application for software anomaly detectors [6, 44, 56, 62, 73]. Benso et al developed a software-level analytical model to predict which application variables are critical for SDCs and demonstrated its accuracy with small benchmarks [6]. Other work developed models to depict how hardware faults lead to application crashes [44, 56, 62] and derived hardware-level detectors [44] and application-level detectors [56, 62] from such models.The application-level detectors were chosen based on metrics such as fanout, lifetime, etc. that signify the criticality of the value to propagate the fault. By

selectively protecting these variables, they demonstrated that the detection latency and SDCs may be reduced. The PVF metric also studied fault propagation through a program to remove the microarchitecture-dependent components from AVF [73]. The PVF metric predicts when a fault may be masked by the program but does not distinguish faults that are detected from those that are SDCs.

Although this dissertation also studies program properties to understand how faults propagate through an application, it differs from the above works its goal is to reduce SDCs from software anomaly detectors; the focus of much of the prior work was on fault detection [44, 56, 62] or to understanding fault masking [73]. The only other work that tried to identify SDCs was the work by Benso et al. [6] which studied small benchmarks that had 100s of variables and for for 1000s of instructions; we use real-world SPEC workloads that contain millions of values and execute for billions of instructions. At such a scale, the analytical models proposed by Benso et al are unusable, making our evaluations more realistic.

2.5 Other Related Work

Once a software anomaly is detected, the source of the fault needs to be diagnosed and repaired before proceeding with recovery. We leverage the SWAT diagnosis module for fault diagnosis [28, 34], and other prior work for repair [9, 59, 65] and focus here on the fault detection and recovery components of SWAT.

Chapter 3

Fault Detection

Fault detection is the cornerstone of any fault tolerance solution. Its purpose is to ensure that faulty execution is promptly flagged and that the appropriate action to restore fault-free execution is initiated. The focus of this thesis is on studying faults in the field. In such a mode of operation, faults are rare and the common mode of execution is fault-free. Therefore, the "always-on" fault detection module should be designed to incur minimal overheads during fault-free execution.

Traditional systems have used heavy amounts of redundancy (in space and time) for fault detection. For example, the IBM-G5 system employed dual-module redundant hardware for fault detection, incurring at least a 100% overhead in core area from the duplicate hardware [72]. Such solutions are, however, no longer applicable owing to the high overheads that they impart on fault-free execution.

SWAT uses a collection of simple low-cost detectors to detect (permanent and transient) hardware faults. These detectors monitor the execution of the software and look for anomalous software execution to indicate the presence of a fault in the underlying hardware. This chapter presents these low-cost software anomaly detectors, along with an evaluation of their efficacy to detect hardware faults.

3.1 Low-Cost Software Anomaly Detectors

SWAT employs the following low-cost detectors that monitor anomalous behavior of both the application and the OS for fault detection. A strategy to implement each detector such that it incurs low hardware cost is also discussed below.

3.1.1 Fatal Traps

This detector monitors for illegal software operations identified by traps that are not seen during normal software operations. SWAT designates the following traps as *fatal* – Divide by Zero, Misaligned Memory Access (in SPARC), Data Access Exception, Illegal Instruction, RED state (Recover Error and Debug, thrown on excessively nested traps), and Watchdog Timer (thrown when no instruction retires in the last 2^{15} ticks). This detector can be implemented at near-zero cost by monitoring the trap state from the hardware trap handling unit. Existing performance counter that monitor such events from the trap-handler may also be used for this purpose.

3.1.2 Hangs

An application or an OS hang is an indication of software misbehavior. SWAT employs a low-cost heuristic hang detector that is implemented in hardware to monitor the frequency of retiring branches and to identify hangs. If a branch executes more frequently than a pre-defined threshold, a hang is flagged. Previous work has proposed hardware implementations for detecting hangs while incurring low area and power overheads [52]. Such implementations may also be leveraged for hang detection.

3.1.3 Kernel Panics

Modern operating systems employ several software-level checks to detect erroneous kernel execution. If such checks fail, the execution of the kernel is frozen to prevent further fault propagation and the kernel is forced to enter a *panic* state. SWAT thus monitors for when the kernel panics, and identifies faults that have corrupted system state. Implementing this detector requires (existing) support from the OS to declare the location of the panic handler (set of PCs of the function) that SWAT can then monitor.

3.1.4 High OS

Typical OS invocations take 10s to 100s of instructions to complete in order to reduce the amount of time spent in the OS (interrupts and system calls for I/O operations are exceptions). Thus, an

abnormally high amount of contiguous activity (10s of 1000s of instructions) represents anomalous software behavior, indicating an underlying fault. SWAT uses performance counters that exist in today's hardware to monitor such information and to identify *High OS* activity.

3.1.5 Application Aborts

Applications that perform illegal operations (such as accessing addresses on invalid pages, or writing to pages without sufficient permissions, etc.) may be terminated by the OS by signaling them [43]. SWAT thus uses application aborts as a direct indicator of anomalous software behavior. Identifying such *App-Aborts* require that the OS be modified to inform the hardware of the signaling mechanisms, either through debug registers or through other schemes. Currently, SWAT monitors application aborts for non-daemon applications by monitoring when the system idle loops in the OS and inferring that the application was terminated (since daemons spend a lot of time idling in the OS, this indirect method would not work for such applications).

3.1.6 Address Out-of-Bounds

Faults that affect data values are known to cause violations in addresses accessed by loads and stores. Software bugs and security violations also cause variations in address values, motivating much prior work that has proposed out-of-bounds detectors for corrupted addresses [3, 19, 20, 50]. Such detectors use sophisticated monitors that track the precise objects being manipulated and perform validity checks on their boundaries. Hardware faults, however, result in more obvious address corruptions such as unallocated addresses on valid pages, or invalid addresses. Thus, a simple detector that identifies legal address limits is sufficient to detect the perturbations caused by hardware faults.

SWAT thus deploys a low-cost out-of-bounds detector that leverages software support to identify such out-of-bounds accesses. Figure 3.1 shows the address space of an application within the OpenSolaris architecture, and how the software can assist the hardware in tracking the limits. All application data loads and stores must access addresses in the *globals address space*, the *heap*, or on the *stack*. The sizes of the globals and static variables are known at compile time. The changes to

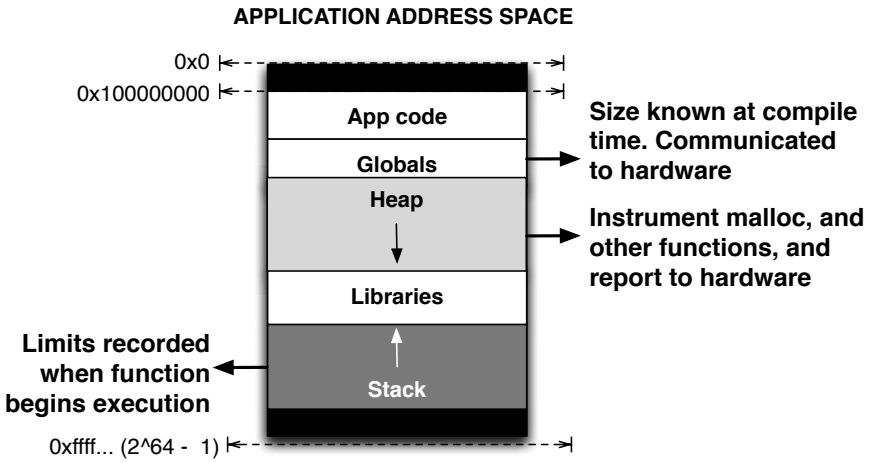


Figure 3.1: Identifying the limits of the globals address space, heap and the stack to detect out-of-bounds accesses. The software communicates the limits to hardware which enforces the checks.

the heap boundaries are identified by instrumenting library function calls that grow the heap. The current top of the stack may be identified by instrumenting call sites in the application to identify the size of the activation record used to setup a function call that grows the stack. Each of these limits are then communicated to the hardware using predefined registers against which all loads and store access are checked for compliance. Since these checks are off the critical path, this detector imparts negligible performance overheads to fault-free execution. The incurred hardware overhead is minimal as only a few registers and some comparison circuitry to check the generated addresses are required. (We also implemented a more sophisticated detector that monitors precise object boundaries, similar to HardBound [19], but found that it improves our results only marginally.)

These software anomaly detectors have also been augmented with software-level invariants mined using the compiler in the iSWAT framework [66]. iSWAT significantly reduces the fraction of faults that escape detection, with low rates for false positives, reducing the overall SDC rate. This thesis does not study these detectors for fault detection due to lack of sufficient software support to instrument the application binary (we couldn't find a working compiler back-end for the SPARC V9 ISA).

The occurrence of any of the above events cedes control to a thin layer of SWAT firmware (without making the event visible to the user), which initiates diagnosis and recovery. Since each of

Per-core parameters	
Frequency	2.0GHz
Fetch/decode/ execute/retire	4 per cycle
Functional units	2 Int add/mul, 1 Int div 2 Load, 2 Store, 1 Br 2 FP add, 1 FP mult 1 FP div/sqrt
Integer FU latencies	1 add, 4 mul, 24 div
FP FU latencies	4 default, 7 mul, 12 div
Reorder buffer size	256
Register file size	256 integer, 256 FP
Load-store queue	64 entries
Memory Hierarchy Parameters	
Data/Instruction L1	64KB
L1 hit latency	2 cycle
L2 size	2MB
L2 hit/miss latency	6/80 cycles

Table 3.1: Parameters of the simulated processor.

these software anomaly detectors monitor events that occur only during anomalous execution and may be implemented with near-zero hardware overheads, fault-free execution is minimally affected.

3.2 Evaluating SWAT Detectors

We evaluate the efficacy of these detectors to detect hardware faults through microarchitecture-level fault injections in a full system simulator. We chose this form of evaluation over hardware fault injections [31], gate-level fault injections [12, 44], and fault injections in FPGA [57] owing to its combination of high controllability and observability at sufficient speed, without a significant compromise in accuracy [33]. We do not perform high-speed mixed-mode fault simulations [33] as we do not have gate-level models for all modules of interest.

3.2.1 Simulation Environment

The simulation is performed by a combination of the Simics full system functional simulator [76] and the Wisconsin GEMS detailed timing models [40]. Within this framework, we simulate a full

Benchmark	Description	Fault-free Output
apache	Provides web pages and files to requesting clients using the HTTP protocol. 4 threads service the requests from a synthetic client driver with 20 threads, obtained from the cURL [15] utility.	Each client thread receives the requested files that are the same as stored on the server.
sshd	Provides files to clients using the SSH protocol. One daemon thread listens to a synthetic client system with 10 threads, and spawns threads with added connections.	Each client thread receives the requested files that are the same as stored on the server.
squid	The server caches remote web pages and services requests from 7 requesting client threads by providing cached pages.	Each client thread receives the requested files from the squid server. The metadata from the server is not considered as part of the output.
mysql	The server performs a sequence of queries provided by a client on over 10,000 inserted entries. The queries consists of 112 select queries with varying conditions, on created tables.	Client records outputs of all queries. It also requests the inserted data for consistency check.

Table 3.2: Description of server workloads, along with their outputs, used in the evaluation. In addition to these workloads, we also evaluate SWAT on all the 16 SPEC CPU 2000 C/C++ codes.

system that implements the SPARC V9 ISA with a modern out-of-order processor a full memory hierarchy, and peripheral I/O devices. The processor and memory are simulated with cycle-accurate GEMS timing models (Table 3.1 gives their configuration) while the rest of the system is simulated functionally.

For workloads, we use 4 I/O intensive server workloads (Table 3.2) and all 16 SPEC CPU 2000 C/C++ codes running on the OpenSolaris operating system. The server workloads are set up in a distributed client-server environment inside Simics using two systems with identical configurations connected by a simulated Ethernet link with a latency of 0.1ms. In these workloads, the server is simulated in detail with the GEMS timing models (Table 3.1) while the client is simulated functionally with Simics.

In this thesis, we restrict our evaluation to faults in systems with single core, and do not explore SWAT in multicore systems (the server workloads that we use are, however, multithreaded). Interested readers are referred to mSWAT which shows that these detectors are also effective in detecting faults in multithreaded workloads running on multicore systems [28].

3.2.2 Fault Model

We inject microarchitecture-level stuck-at-0 and stuck-at-1 permanent faults and transient faults by leveraging the timing-first approach used in the GEMS+Simics infrastructure [42]. 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 between 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 the case that the mismatch occurs because of such an instruction.

We modified this checking mechanism for the purposes of microarchitecture-level fault injection. We inject a fault into the timing simulator's microarchitecture state and track its propagation as the faulty values are read through the system. When a mismatch in the *architecture 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' architecture 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 the faulty execution trace, upholding the timing-first paradigm.

We say an injected fault is *activated* when it results in corrupting the architecture 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).

In each run, we inject one fault in a randomly chosen bit in one of the representative structures of a modern processor in Table 3.3. For each application, we pick 4 base injection points (or *phases*) spaced sufficiently apart in the entire execution, to capture different application behavior. We do not use SimPoints [27] to pick the phases for the SPEC workloads as they were designed for performance evaluations and their relevance to reliability is unknown. For each phase, and each structure, we pick several spatially and temporally random injection points. Since the distribution of faults may not be correlated to the area of a structure, especially for permanent faults, we inject the same number of faults in each structure. For the server workloads, we inject faults only in the

μarch structure	Fault location
Instruction decoder	Input latch of one of the decoders
Integer ALU	Output latch of one of the integer ALUs
Register bus	Bus on reg file write port to the register file
Physical int reg file	An int physical register in the integer register file
Reorder Buffer (ROB)	Entry's source/destination register number of an instr in ROB entry
Reg Alias Table (RAT)	Log→ phys mapping
Address unit (AGEN)	Virtual address generated by the unit
Floating Point Unit	Output latch of one FPU

Table 3.3: Fault injection locations.

server systems to model a situation where the server, serving remote clients, is subjected to faults and is protected by SWAT.

Using this randomized selection, we inject a total of 8960 permanent and 8960 transient faults while running server workloads and an equivalent number while running SPEC workloads. This gives us statistically significant samples for our results. To attain this number of injections, we select 40 random injection points in a server workload phase and inject one stuck-at-0 and one stuck-at-1 faults during the execution (giving a total of 4 applications $\times$ 4 phases $\times$ 7 structures $\times$ 40 random points $\times$ 2 fault models = 8,960 faults). For transient faults in servers, we pick 80 such injection points. Since we simulate faults in 16 benchmarks in SPEC, and only 4 server workloads, we picked 10 such random points for permanent faults and 20 such random points for transient faults in SPEC.

3.2.3 Fault Detection

After fault injection, we simulate the system for 10 million instructions (including application and OS instructions) to identify the faults that are architecturally masked. Once the fault corrupts the architecture state, or is *activated*, we continue timing simulation for 10 million instructions to detect the fault. For the *High-OS* software anomaly detector, we set the threshold at 20,000 contiguous OS instructions for the SPEC workloads (based on fault-free profiles) and disabled this detector for the server workloads as these daemons execute a lot of idle instructions in the OS while waiting for client requests. The thresholds of the Hang detector are set at 0.1% instructions for server workloads and 1% of instructions for SPEC based on fault-free profiles in order to avoid false positives. If an

activated fault is not detected in this window, we functionally simulate the system to completion without the fault to identify faults masked by the application (i.e., when the final output is the same as the fault-free run), and faults detected at longer latencies. The remaining faults corrupt the output of the application and are classified as *Potential-SDCs*.

3.2.4 Metrics for Fault Detection

We use two metrics to evaluate the efficacy of the detectors to detect hardware faults – Potential SDC rate, and detection latency.

The *Potential SDC rate* is defined as the fraction of injected faults that are activated, escape fault detection, and corrupt the output of the application. Since these faults constitute instances when software anomaly detection fails, anomaly detectors strive to reduce this rate. Further, since some such corruptions may be tolerated by the application, only a fraction of these would constitute true SDCs. (Chapter 4 discusses this distinction in more detail.)

The *detection latency* for the detected faults is defined as the latency between activation (architecture state corruption) and fault detection. The detection latency has a direct bearing on the efficacy and overheads of fault recovery and is hence an important metric to study for any fault detection scheme. (Chapter 4 revisits this definition and presents a more recover-oriented definition of the detection latency.)

The large number of injections across the different workloads (8960 per fault model for each class of workloads) results in low error bars at high confidence intervals for these metrics. At a 95% confidence interval, the measured potential SDC rate has an error bar of under 0.1% for faults in each fault model (transient and permanent hardware faults) injected in each class of workloads (SPEC and server workloads). For the detection latency, the fraction of faults detected in under 10M instructions also has a low error bar of under 0.5% at a confidence interval of 95%, making our results statistically significant.

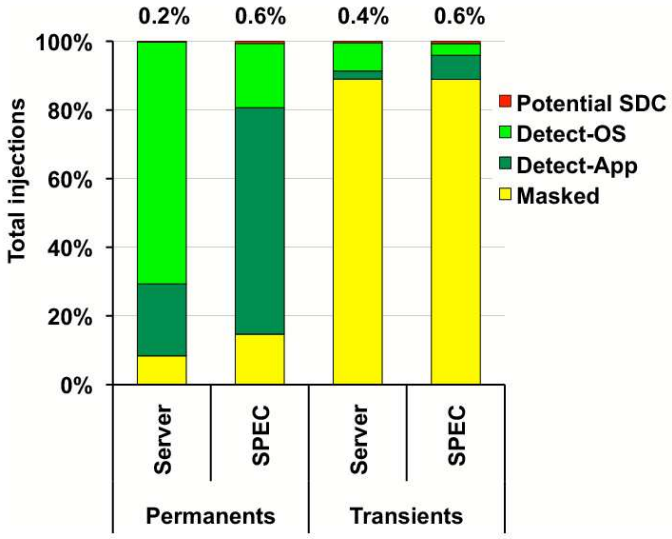


Figure 3.2: Potential SDC rates from permanent and transient faults injected into non-FP units in server and SPEC workloads. The low rates show that the SWAT detectors are highly effective in detecting hardware faults.

3.3 Results

3.3.1 Potential SDC rate

Overall Results

Figure 3.2 shows the overall outcome of (a) permanent and (b) transient faults injected into non-FPU structures for the server and SPEC workloads. For faults in each type of workload, the stacks show the fraction of injected faults that are *Masked*, detected by anomaly detectors from the application (*Detect-App*), detected by anomaly detectors from the OS (*Detect-OS*), and that result in *Potential SDC*. Since the server workloads do not use the FP unit, we do not inject faults into the FPU for the server workloads and drop the results from the injections into the FPU for the SPEC workloads as well, in order to keep the comparison uniform. (We discuss results from the FPU separately.) The numbers on top of each bar shows the potential SDC rate for faults in the given workloads.

From the figure, we see that the Potential SDC rate is $< 0.6\%$ for the injected permanent and transient faults in SPEC and server workloads. The low rates for the permanent faults shows that a large fraction of such faults affect the software and cause anomalous execution, which is monitored

by the SWAT detectors. The transient faults in Figure 3.2, demonstrate a high rate of masking (85% and 88% of the transient faults injected in the server and SPEC workloads are respectively masked). This high masking rate is because of microarchitecture-level masking, faults in dead values, and logical masking; these results are consistent with previous work [61, 78].

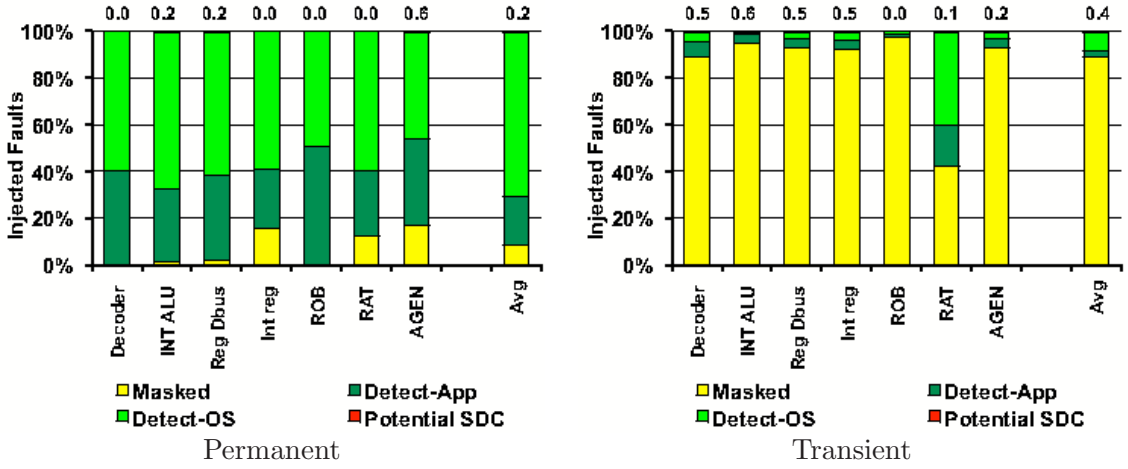
The figure also shows that a large fraction of detections come from the OS – Detect-OS accounts for 77% and 23% of the faults detected in the server and SPEC workloads, respectively. Since the simulated system has a software managed TLB, the OS is invoked to handle TLB misses caused by the fault. This may result in the OS further activating the fault and undergoing anomalous execution. This shows that simulating the operating system is important when studying the impact of hardware faults.

The SWAT detectors are therefore effective in detecting hardware faults, resulting in low potential SDC rates. Further, the low overheads (in performance, power, and area) associated with these detectors makes them an attractive option for low-cost fault detection in future systems.

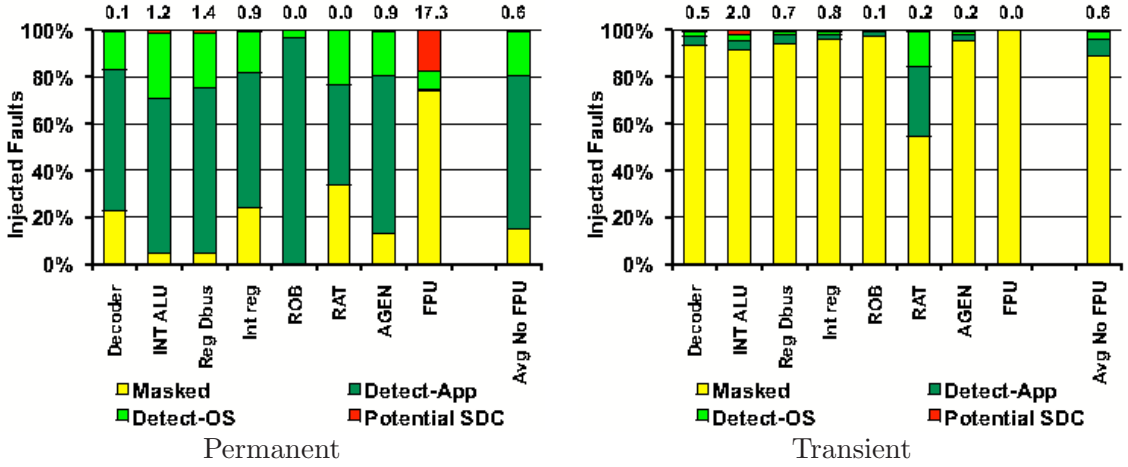
Per-Structure Results

Figure 3.3 further breaks these results down on a per-structure basis for permanent (left-side) and transient (right-side) faults in (a) server, and (b) SPEC workloads. (The outcome of faults injected into the FPU for SPEC workloads is also shown.) In each graph, for each structure, the stacks in the bars correspond to faults that are *Masked*, detected by the application *Detect-App*, detected by the OS *Detect-OS*, and *Potential SDCs*. The number on top of each bar shows the potential SDC rate for faults in that structure. The *Avg* bar in each graph shows the aggregate results for faults in all structures excluding the FPU (same as the bars in Figure 3.2).

From the figures we see that the potential SDC rates of the structures that directly influence the control flow of the application (Decoder, ROB, and RAT) is lower than 0.5% across the all the bars. This is because the SWAT detectors are adept at detecting faults that affect control flow. Faults in structures that affect data-flow (Int ALU, Reg Dbus, Int Reg, AGEN, and FPU) tend to have higher rates of potential SDCs than the former 3 structures. However, barring permanent faults in the FPU, even these structures have rates of under 2.0%. This is because a large fraction of the



(a) Server workloads



(b) SPEC workloads

Figure 3.3: Per-structure breakdown of outcomes from permanent and transient faults in (a) server and (b) SPEC workloads. While SWAT is effective for most structures, yielding low SDC rates. Faults in structures that operate on purely data-only values, such as the FPU, cause higher rates of potential-SDCs, warranting additional support for software anomaly detection.

values corrupted from faults in these structures also affect values that are used in control decisions and/or memory addresses, resulting in anomalies that the SWAT detectors monitor. (Chapter 6 builds this intuition further.) The SWAT detectors are therefore effective to detect faults in most hardware structures.

We now turn our focus to the permanent faults injected in the FPU for SPEC workloads (left-side graph of Figure 3.3(b)). Since 12 of the 16 SPEC C/C++ workloads are Integer workloads that sparingly use the FPU, 74.4% of the injected permanent faults (and all the injected transient faults) in this structure are masked. Only a small fraction of the unmasked faults are detected by the SWAT detectors, resulting in a high potential SDC rate of 17.3%. Since the floating point data values that get corrupted from these faults are primarily used only for pure data computations, without affecting control decisions or memory addresses, they do not result in software-visible anomalies. The SWAT detectors discussed here are therefore insufficient to detect faults in such structures, warranting additional support. Software-level invariants that target data values [66] and circuit-level BIST techniques [54] may be used to protect the software from faults in such structures. Chapter 4 discusses these potential SDCs further and demonstrates that a large fraction of these faults may actually be tolerated by the application.

3.3.2 Detection Latency

Another important aspect of fault detection is the latency at which a fault that corrupts the architecture state is detected. Since the detection latency directly governs fault recoverability and the overheads from operations related to fault recovery (such as checkpointing and output buffering), it is important that faults are detected at short latency to keep these overheads low.

Overall Results

Figure 3.4 shows the cumulative percentage for detected permanent and transient faults across all structures (without the FPU) in (a) server and (b) SPEC workloads. The faults are categorized in to the following latency bins – $< 10K$, $< 100K$, $< 1M$, $< 10M$, and $> 10M$ instructions. Faults detected at $> 10M$ are detected in the functional simulation mode after the fault is turned off.

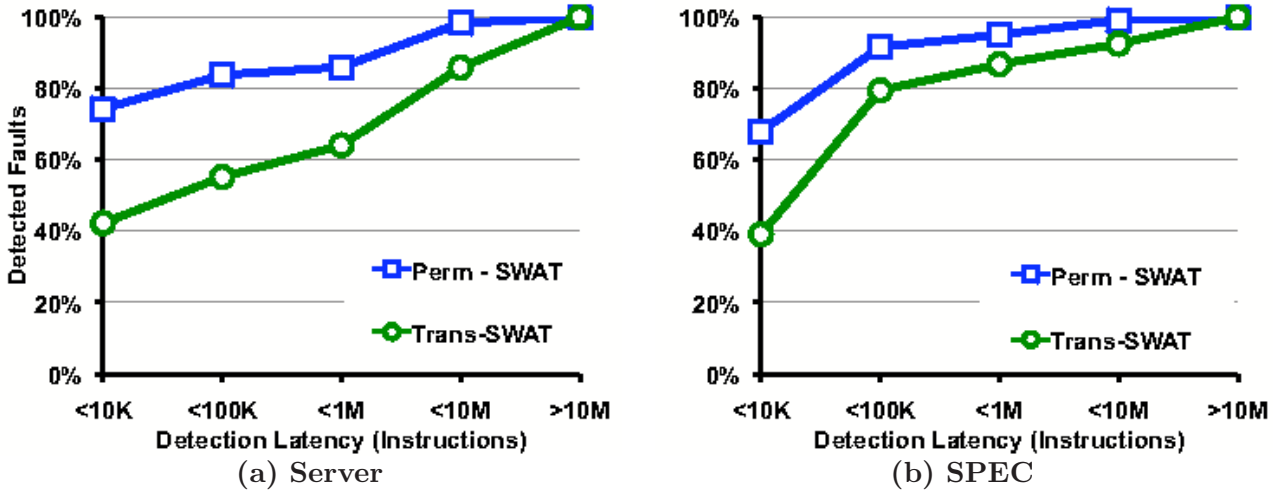


Figure 3.4: Detection latency for detected non-FPU permanent and transient faults in (a) Server and (b) SPEC workloads. Over 98% of the detected faults are detected at a latency of $< 10M$ instructions, making then recoverable with hardware checkpointing.

The figures show that at a given detection latency, more permanent faults, than transient faults, are detected. This is because permanent faults may be activated by multiple instructions, corrupting multiple values in the application, and making them more visible to the software anomaly detectors. Transient faults, on the other hand, are activated only once, making them harder to detect.

At a short latency of 100K instructions, 81% of the detected faults (both permanent and transient faults) are detected. At this latency, the overheads from the operations for fault recovery (checkpointing and output buffering, discussed in Chapter 5) are minimal, making the SWAT detector effective in enabling low cost recovery solutions.

Existing hardware checkpointing scheme, like SafetyNet [71] and ReVive [60], claim to support recovery at latencies of 10s of milliseconds, which corresponds to approximately 10M instructions in a modern processor (assuming a 1GHz processor with an IPC of 1). At this latency, Figure 3.4 shows that 99% of the detected permanent faults and 93% of the detected transient faults are detected, making such faults amenable to hardware recovery. (Chapter 5 further explores the validity of these claims in the presence of device I/O.)

Faults detected at latencies longer than those handled by hardware checkpointing schemes may require additional support from software [10, 16, 25, 36]. Such hardware-software cooperative tech-

niques are emerging as low-cost implementations of fault recovery for future systems and require further exploration which is beyond the scope of this thesis.

Per-Structure Results

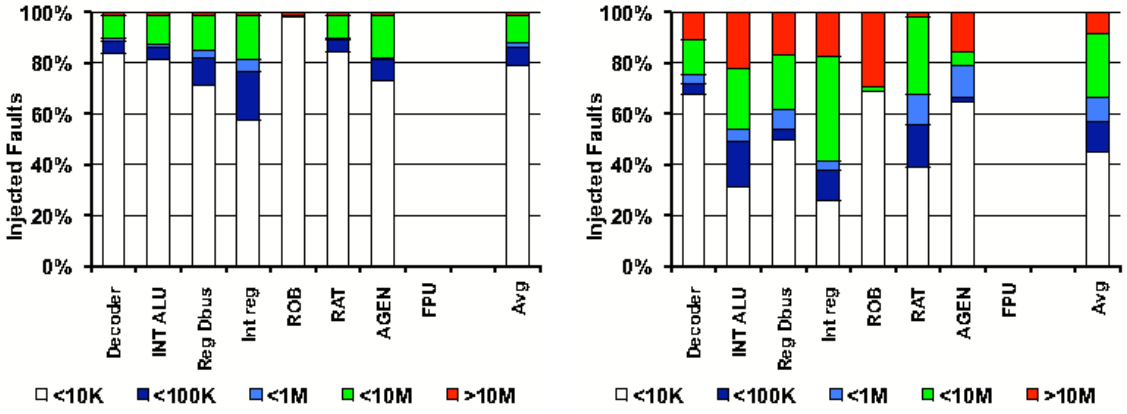
Figure 3.5 categorizes the detection latencies on a per-structure basis. For permanent faults (left-side graphs) and transient faults (right-side graphs) in (a) server and (b) SPEC workloads, each bar bins the detection latencies of the detected faults in to the following bins – $< 10K$, $< 100K$, $< 1M$, $< 10M$, and $> 10M$. The *Avg* bars shows the cumulative results for faults injected into the non-FPU units. Since the server workloads do not use the FPU, no FPU faults were injected (and hence, none were detected). For the SPEC workloads, while some permanent FPU faults were detected, all the injected FPU transient faults were either masked or caused SDCs, giving it zero detections.

Comparing the permanent and transient faults detected across both sets of workloads, we see that the transient faults incur longer detection latency for reasons previously described. Consistent with the structure-specific results for the potential SDC rates (Figure 3.3), we see that the detection latencies for faults in structures that directly affect control operations (Decoder, ROB, and RAT) to be lower than the detection latencies for the other structures.

3.3.3 Contributions from Each Software Anomaly Detector

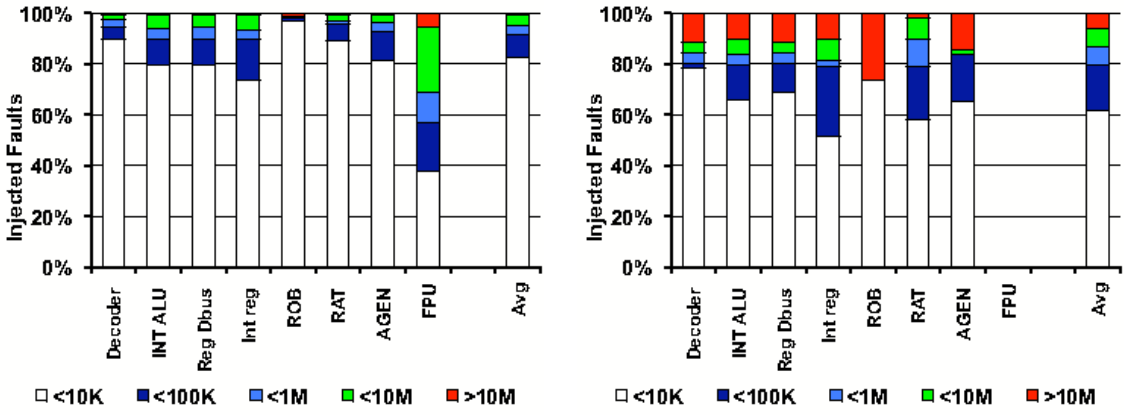
We now study the efficacy of each software anomaly detector employed by SWAT to detect permanent and transient faults. Figure 3.6 shows the distribution of the faults detected per detector (aggregated across both permanent and transient faults) for (a) server and (b) SPEC workloads. The numbers next to each sector gives the percentage of detected faults that are covered by that detector.

Since the High-OS and App-Abort detectors are not used in the server workloads, they do not feature in the chart in Figure 3.6(a). Of the remaining software anomaly detectors, Fatal-Traps account for just under half the detections, making them the most effective detectors for server workloads. Further, since these workloads serve random client requests, they do not have regular



Permanent Transient

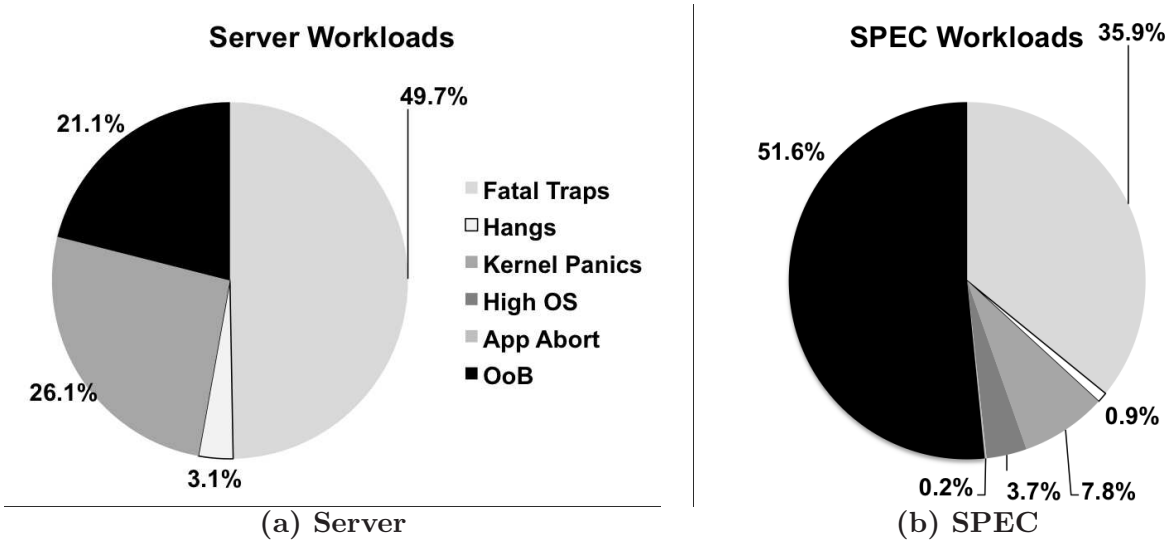
(a) Server workloads



Permanent Transient

(b) SPEC workloads

Figure 3.5: Per-structure breakdown of the detection latency of permanent and transient faults in (a) server and (b) SPEC workloads.



(a) Server (b) SPEC

Figure 3.6: Contribution of each detector towards detecting permanent and transient faults in (a) server and (b) SPEC workloads.

memory access patterns making the out-of-bounds detector less effective for faults in server workloads (only 21% of the faults detected in server workloads are covered by this detector, compared to a much higher 51% for SPEC).

In contrast, for faults in the SPEC workloads (Figure 3.6(b)), the Out-of-Bounds detector accounts for over 51% of the detections owing to the regular memory access patterns of these workloads, making it the most effective detector. Fatal-Traps account for the next largest slice of the pie, detecting 36% of the detected faults. Further, there are small contributions from each software anomaly detector, making each detector important to achieve the low potential SDC rates previously demonstrated.

3.3.4 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

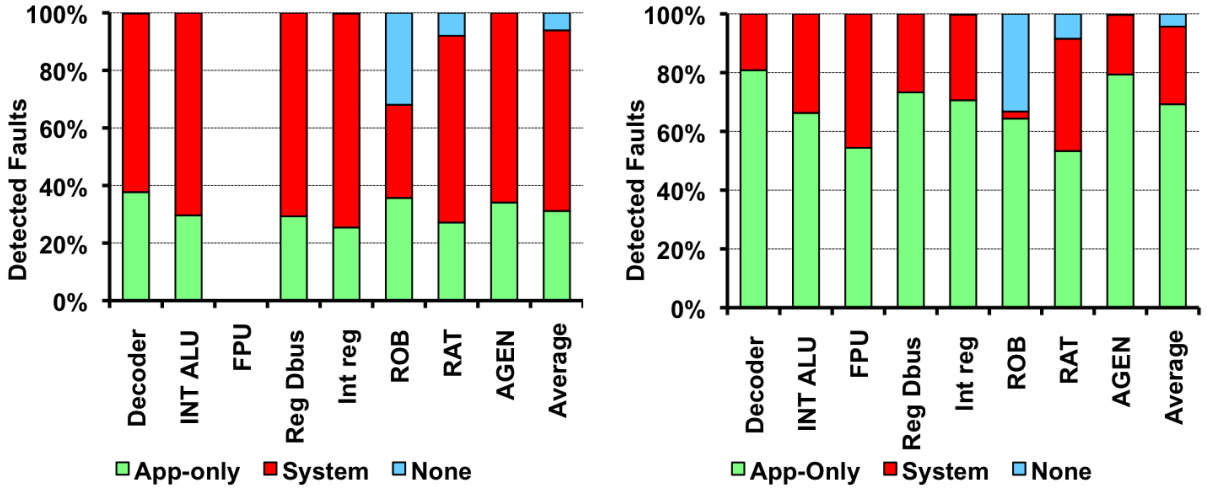


Figure 3.7: Software components corrupted for faults detected in (a) server and (b) SPEC workloads. System state is corrupted in 46% of the faults detected, making it important to restore the system state during fault recovery.

only for a virtual machine approach so far [22] and the feasibility of hardware checkpointing would depend on detection latency.

We note that whether the state of the application or the OS was corrupted is not necessarily correlated with whether the fault was detected from the application or from the OS. A fault could be detected in the OS 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.

Figure 3.7 shows the components of the software that are corrupted for faults detected in each structure in the (a) server and (b) SPEC workloads (the results from the permanent and transient faults are aggregated for brevity). In each bar, the faults detected in each unit are divided into three categories – *App-only* shows the fraction of detected faults in which only the state of the application is corrupted, *System* shows the fraction in which the system state (and potentially the state of the application) is corrupted, and *None* shows the fraction in which the fault is detected before any architecture state is corrupted. (No faults were injected into the FPU for the server workloads, giving an empty bar for the FPU in Figure 3.7(a).)

The figures show that the system state (and potentially also the state of the application) is

corrupted before the fault is detected for a large fraction of the detected faults (62% and 26% of the faults detected in the server and SPEC workloads fall under this category, respectively). The large difference between the fractions for the server and SPEC workloads stem from the fact that the out-of-bounds detector detects many of the faults in the SPEC workload before they corrupt the system state; for the server workloads, the OS is invoked in the simulated system to handle the consequent TLB miss, which further activates the fault corrupting the state of the system. This motivates the exploration of checkpointing the OS and/or fault-tolerant strategies within the OS.

Additionally, there are a few detected fault cases where neither the application nor the OS state is corrupted (32% of detected faults in the ROB and 8% in the RAT across both sets of 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).

3.4 Summary and Implications

This chapter presented a collection of simple low-cost detectors to detect both permanent and transient hardware faults. Since the detectors monitor anomalous software behavior, they are oblivious to the root-cause of the fault, reducing their overheads during fault-free execution.

Our evaluation showed that these detectors are highly effective in detecting permanent and transient faults in compute-intensive and I/O-intensive workloads. In particular, a large fraction of the injected permanent and transient faults in both server and SPEC workloads are detected, resulting in a low potential SDC rate. Further, these faults are detected at sufficiently short latencies that enables recovery with hardware support. However, faults in certain structures that affect purely

data computations, such as the FPU, may require additional support for detection as SWAT incurs a high potential SDC rate for faults in these structures. The results also show that a large fraction of faults are detected through software anomalies from the OS, making system recovery important.

These results have far-reaching implications on the design of resilient systems. First, it demonstrates that identifying hardware faults by observing anomalous software execution is feasible. Since the hardware faults that matter are those that affect software execution, this strategy for detection seems an viable choice for future systems. Second, it demonstrates that the proposed low-cost detectors achieve high reliability targets (analogous to low SDC rates), precluding the need for expensive solutions that involve excessive amounts of redundancy. Finally, while the detectors presented here already achieve low SDC rates, they can be tailored to suit the needs of the application; higher reliability targets may be achieved by using sophisticated detection schemes that may incur higher overheads. For example, the software-level invariant detectors proposed by iSWAT detect several faults that escape these base-line SWAT detectors, but do so at marginally higher performance overheads [66]. Such trade-offs between resiliency and performance/power are expected to become increasingly important in future systems.

Chapter 4

Application Aware Metrics for SWAT

SWAT, like other symptom detectors, identifies the presence of an underlying hardware fault by observing anomalous software execution. By virtue of such a detection strategy, SWAT successfully ignores faults that are masked by higher levels of the hardware and by software as they do not lead to anomalous software execution. While the faults that escape detection affect the outputs produced by the software, certain classes of applications may be able to tolerate even such faults [17, 28, 36]. This is commonly known as application-level fault tolerance and such applications are said to perform *soft-computations* [36]. Common examples of soft-computations include image processing, video compression/decompression, optimization algorithms that may have multiple solutions at the same cost, etc. Since such faults do not result in anomalous execution, they cannot (and arguably should not) be detected through software anomaly detection.

In this chapter, we consider these notions of application-level fault tolerance and revisit the SWAT detectors. In particular, we take a closer look at the potential SDCs and classify which of these may result in true SDCs that produce unacceptable application outputs. We also revisit the detection latency and understand the impact of application-aware metrics on the detection latency and on fault recovery.

4.1 Application Aware Silent Data Corruptions

Faults that escape detection and produce outputs different from the fault-free outputs are traditionally classified as SDCs. However, certain applications may be able to tolerate departures from the fault-free output. If such departures do not flag anomalous behavior, then the SWAT detectors cannot (and arguably should not) be expected to detect them, and the faults should not be classified

as an SDC.

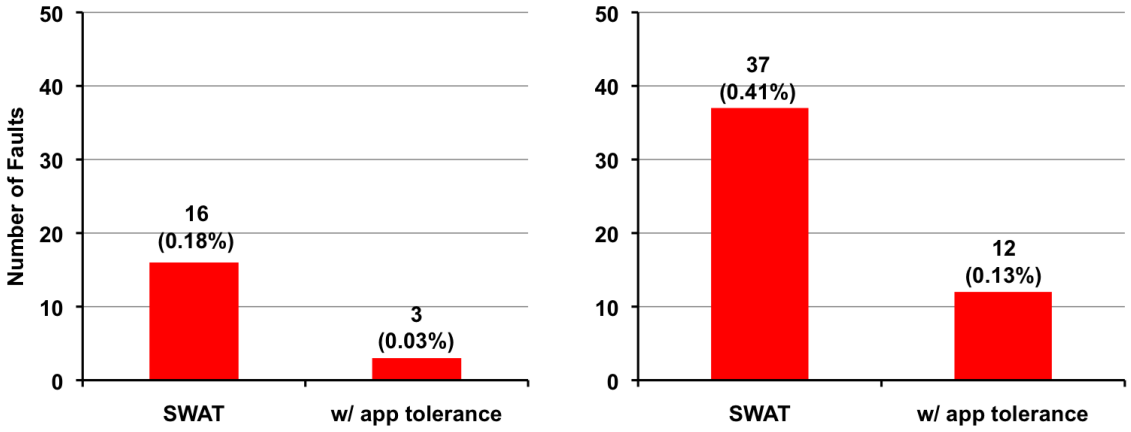
4.1.1 I/O Intensive Distributed Client-Server Workloads

Distributed client-server applications are typically deployed in distributed environments where the client and server may be present at different physical locations. Network interruptions and link failures are common in such distributed systems. Distributed applications that extensively use the network are therefore written to work even in the presence of such failures, making them fault tolerant by design. A typical mechanism used by such applications when a transaction to communicate with the server fails is for the client to retry the transaction (maybe with with back-off). Thus, if the fault corrupts the server thread that is handling a client and sends no response within a fixed interval, the client may retry the request which is typically serviced by a new server thread. Thus, the effect of the fault is masked by the application, not resulting in an SDC.

Table 3.2 gave a description of the server workloads that we use, along with a brief description of its fault-free output. Each of these applications is a distributed client-server application where the fault is simulated on the server. Thus, if an undetected fault results in a lost connection from the client and a failed request, the client can retry the request, depending on the type of the application. If the retry is successful, then the application is said to have tolerated the fault.

Figure 4.1 shows the impact of the application-driven analysis on the potential-SDCs for server workloads. The height of each bar and the number at the top shows the remaining potential-SDCs in absolute terms; the number in parenthesis shows potential-SDCs as a percentage of total injected faults. In our experiments, we explicitly added client-side retries to only the *sshd* workload as a proof-of-concept of the retry behavior. Based on the impressive success of these retries, we classify those faults in the other workloads that also lead to detecting a failed connection with the server (identified through console logs) as tolerated by the application on retry.

The figures show that application-driven analysis has a dramatic effect on the potential SDCs in the server workloads (bars labeled *w/app tolerated*). We find that the simple mechanism of retrying a request on a lost connection seen at the client side eliminates most of the potential-SDCs remaining with the out-of-bounds detector – 81% of the permanent and 67% of the transient faults



(a) Permanent Faults (b) Transient Faults

Figure 4.1: Reduction of potential SDCs in server workloads with application-driven analysis for (a) permanent and (b) transient faults. The numbers at the top of each bar are the number of potential-SDCs in absolute terms and as a percentage of injected faults (in parenthesis). Client-side retries reduce potential-SDCs in server workloads significantly.

that were originally classified as potential SDCs from the server workloads are now tolerated by the application. This shows that considering application-level fault tolerance is important to accurately classify the true SDC rate from real-world workloads.

After incorporating client-side retries, we are left with only 3 permanent faults (0.03% of total injected) and 12 transient faults (0.13% of total injected) as SDCs. Of these 15 faults, only 3 are genuine SDCs – one from *squid* with an error in the a transferred web stream, and 2 from *mysql* with incorrect employee numbers. The remaining are errors have clearly software visible software anomalies – 7 have missing transactions because the connection with the server was prematurely terminated, 2 have invalid entries in the location field of a database in the *mysql* workload, 1 case with the *mysql* workload has an employee number of 0, and two have negative values for salary. Server applications often have integrity checks and assertions built into them for software bug detection – these could potentially be leveraged to detect the above SDCs as well.

4.1.2 Compute Intensive SPEC Workloads

For other types of applications, although the fault may propagate to generated outputs, the application may be able to tolerate such differences in outputs, making them inherently fault-tolerant. For

example, a noisy image output may be an acceptable output of an image rendering engine as long as the Signal-to-Noise Ratio (SNR) of the output image is above a certain threshold. Such applications are said to perform *soft computations* that can tolerate certain faults in their outputs [36].

Expanding on previous work that explores application-level fault tolerance [17, 36], we analyze all the SPEC C/C++ workloads to understand their ability to tolerate faults. Although these workloads have been traditionally thought of as tolerating zero errors in output, an analysis of the applications, and observations from previous work [36], revealed that several SPEC workloads perform soft computations. Table 4.1 gives a description of the fault-free outputs of the C/C++ SPEC CPU 2000 benchmarks that tolerate errors in output values. For these applications, we evaluate the output produced in the presence of a fault based on two aspects – consistency and quality, as defined in Table 4.1. The remaining 4 Integer benchmarks, namely *197.parser*, *253.perlbmk*, *254.gap*, and *255.vortex*, do not tolerate any differences in their outputs, and any departures of the output from the fault-free output are considered to be SDCs.

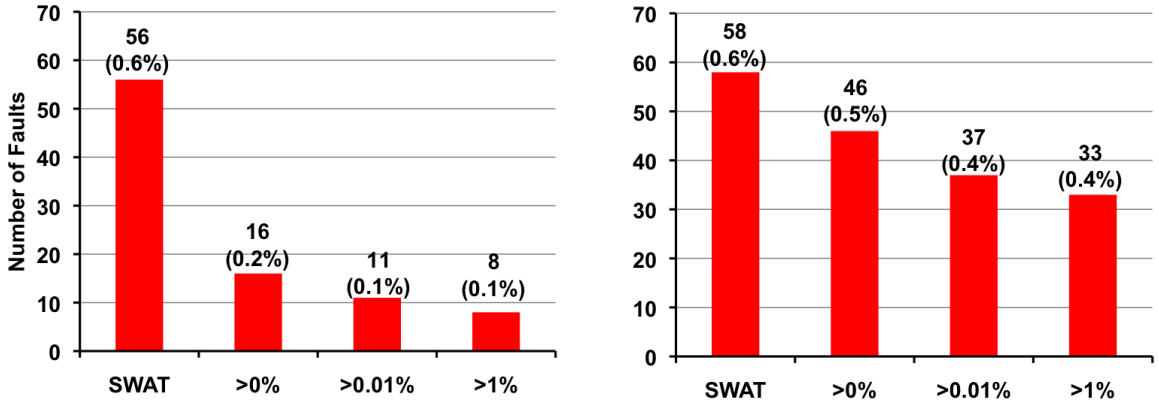
For the benchmarks shown in Table 4.1, even though the faulty output may be different from the fault-free output, it may be acceptable as long as the output passes the consistency check of the benchmark. The quality of the output can then be judged on application-specific thresholds for acceptable error in outputs (relative to the fault-free output). For certain benchmarks that produce multimedia outputs, like *252.eon* and *177.mesa* in Table 4.1, the consistency of the output is defined by its quality (a low quality output may be considered inconsistent).

Results from non-FPU faults

We apply this notion of application-level fault tolerance to categorize the undetected faults into those that are tolerated by the application, and those that may result in SDCs. Figure 4.2 shows these results for faults in non-FPU units (we discuss faults in the FPU separately). It shows the number of potential-SDCs when an output quality degradation of $> 0\%$, $> 0.01\%$, $> 0.1\%$, and $> 1\%$ relative to the fault-free output is deemed acceptable. Thus, a potential-SDC case where the output is different but of the same quality as the fault-free output (i.e., degradation is 0%) is counted as masked for all of these bars (and not included in the potential-SDC count). Potential-

Benchmark	Category	Fault-free output	Consistent output	Output quality
Integer benchmarks				
164.gzip	Compression	Compressed file	Lossless compressed output	Error in compression ratio
175.vpr	FPGA circuit placement and routing	Detailed placement and cost information	Valid placement satisfying constraints	Absolute error in cost
176.gcc	C programming language compiler	Assembly program for the Morotola 88100 processor	Valid assembly with same output	Performance
181.mcf	Combinatorial optimization of a scheduling problem	Schedule with minimal cost	Valid schedule	Absolute error in cost of schedule
186.crafty	Chess simulation	Game outcome with number of moves to mate	Valid game outcome	Error in number of moves
252.eon	Computer visualization	Image file in PPM format	PPM image file	Peak SNR
256.bzip2	Compression	Compressed file	Lossless compressed file	Error in compression ratio
300.twolf	Place and route simulator	Detailed placement information and cost	Valid placement satisfying constraints	Absolute error in cost
Floating point benchmarks				
177.mesa	3-D graphics library	2D image file in PNG format	PNG image file	Peak SNR
179.art	Image recognition/Neural networks	Field ((x,y) coordinates) of recognized objects	Successful recognition	RMS error in location
183.equake	Seismic wave propagation simulation	Displacements caused by seismic wave	Successful seismic simulation	RMS error in displacements
188.ammp	Computational chemistry	Energy of 3 molecules	3 energy values, one for each molecule	RMS error in energy values

Table 4.1: Inherent fault tolerance of SPEC C/C++workloads. The errors under output quality refer to the difference from the fault-free output. The following 4 of the 16 SPEC CPU 2000 C/C++ workloads do not tolerate error in their output and are not listed above– *197.parser*, *253.perlbmk*, *254.gap*, and *255.vortex*. Any departures of the outputs of these applications from their fault-free outputs are classified as SDCs.



(a) Permanent Faults (b) Transient Faults

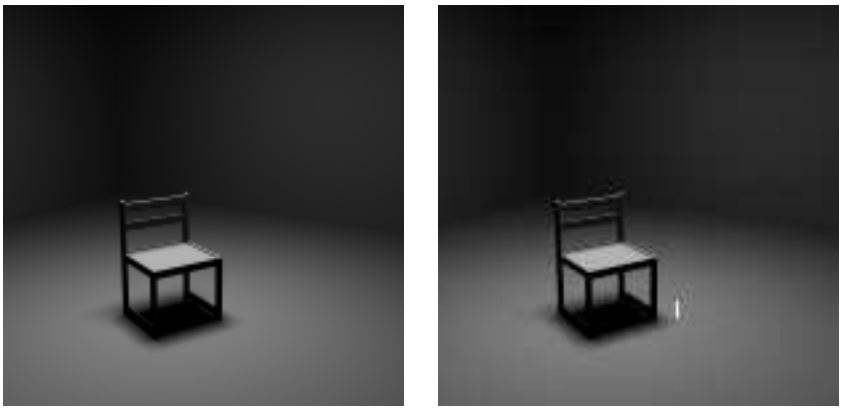
Figure 4.2: Reduction of potential SDCs in SPEC workloads with application-driven analysis for (a) permanent and (b) transient faults. The numbers at the top of each bar are the total number of potential-SDCs in absolute terms and as a percentage of total injected faults (in parenthesis). The bars labeled $> X\%$ show the number of potential-SDCs if quality degradation of $X\%$ is assumed as acceptable, with the baseline and out-of-bounds SWAT detectors. The application-driven analysis shows that many of the potential-SDCs do not affect the quality of the solution.

SDCs from applications that do not have a notion quality degradation (i.e., any departure from fault-free output is deemed as infinite degradation) are included in all of these bars.

From the figures, we see that the 40 of the 56 permanent faults and 12 of the 58 transient faults that were classified as potential SDCs are just different solutions with the same quality as the fault-free outputs (such as a different schedule with the same cost, or a different placement with the same cost, etc.). This shows that simply accepting alternate outputs with no degradation in the output quality reduces the potential SDC rates significantly – the rate drops by 68% for permanent faults, and by 20% for transient faults.

The other bars in Figure 4.2 tolerate an increasing amount of error in the quality of a consistent output. From the figures we see that lowering the threshold for tolerance further reduces the SDC rate seem by the detectors. At a tolerance level of $< 1\%$ for quality, SWAT has just 8 permanent faults and 33 transient faults from SPEC workloads classified as SDCs (0.2% of the injected 17K faults). Application-aware SDC categorization is thus effective in identifying the true SDCs from SWAT and other software anomaly detectors, and accurately measuring their SDC rates.

The 41 faults that are now classified as unacceptable SDCs may be detected with additional



(a) Fault-free output **(b) An SDC**

Figure 4.3: An example of an SDC in the application eon. The small perturbations caused by the faults are not detected by the anomaly detectors employed by SWAT, resulting in small variations in the image output image. Support from the software may be leveraged to detect such faults and lower the SDC rate further.

support from software. An analysis of the corrupted outputs revealed that many of these faults corrupt pure data-values without corrupting much of the control flow of the application. (Chapter 6 builds an intuition behind these hard-to-detect faults further.) In order to visually see such an SDC, Figure 4.3(a) shows the fault-free output of the SPEC application *eon*, while Figure 4.3(b) shows an output that is classified as an SDC. The image in Figure 4.3(b) is classified as an SDC because the small perturbations in the rendering of the image of the chair results in a PSNR of < 50 (the output classified as SDC has a small vertical white line to the right side of the chair; the fault-free output does not have this line). Although we classify this perturbation as an SDC, a less stringent classifier may consider this output to be acceptable. For some outputs that were classified as SDCs, we observed more significant perturbations. In those cases, software-level support, in the form of detectors that monitor such semantics of the application, might be necessary for fault detection. We leave devising and studying such detectors to future work.

Results from FPU faults

Figure 4.4 is similar to Figure 4.2 but shows the effect of application-level fault tolerance for permanent faults injected into the FPU. (We do not study transient faults injected into the FPU as

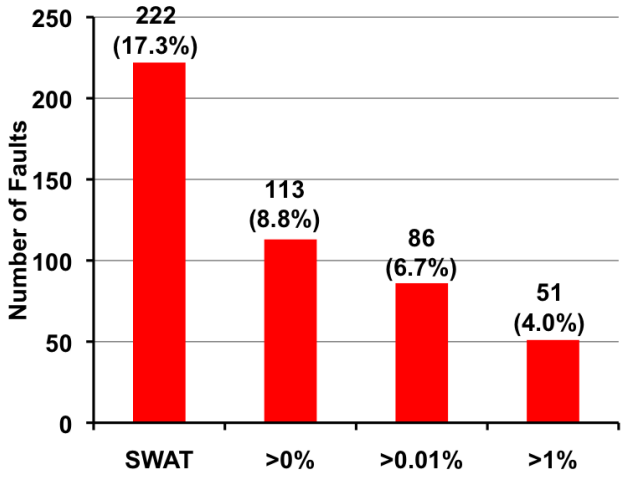


Figure 4.4: Application-aware analysis of permanent faults in the FPU with SPEC workloads. A large fraction of the faults in the FPU that were previously categorized as potential SDCs are tolerated by the SPEC workloads. Additional support is however required to lower the SDC rate for faults in such units that are used for pure data computations.

they were all masked, as shown in Figure 3.3(b).) The results from faults in the FPU are shown separately as previous results on fault detection (Figure 3.3 in Chapter 3) showed that permanent faults in the FPU with SPEC workloads yielded a high potential SDC rate of 17.3%, making it a vulnerability of SWAT. For increasing threshold of quality of the output, the figure shows the number and the fraction of faults injected into the FPU that are classified as SDCs. The numbers for the baseline SWAT system are also shown.

Similar to results on other structures, the number of faults in the FPU that are categorized as potential SDCs reduce significantly as we increase the threshold for acceptable output quality. When only alternate solutions with no degradation of output quality if accepted (the $> 0\%$) bar, the SDC rate in the FPU reduces by 49% to a rate of 8.8%. As we increase the threshold further, even a conservative threshold of 1% is effective in reducing the number of SDCs to a mere 51, resulting in an SDC rate of 4.0% for permanent faults in the FPU. Additional support from the application for FP values, such as using invariants generated by iSWAT [66] and other software-level techniques [37, 80], may be used to reduce this rate even further.

4.2 Application Aware Detection Latency

Traditionally, once the fault propagates to the architectural state recorded in a checkpoint, the system is said to be no longer recoverable as the architectural state corruption remains even after the system rolls back to the checkpoint. Thus, detection latency has been defined as the latency from architecture state corruption to detection (the *Hard-Latency*) and is assumed to govern the recoverability of the fault – if the last checkpoint available for fault recovery was taken within this latency, all checkpoints are corrupted, making the system unrecoverable. Such faults are traditionally considered to be Detected Unrecoverable Errors or DUEs [48].

In reality, however, recoverability is governed by when the fault corrupts the execution of the software and makes it unrecoverable. Faults that remain in the architecture state even after rollback may be subsequently masked by the software, resulting in successful recovery. Consider the fault-free execution shown in Figure 4.5(a), with checkpoints taken at T_1, T_2, and T_3. If the fault affects the values of `a` and `b` as shown in Figure 4.5(b), the checkpoint at T_1 records fault-free architecture state, while those at T_2 and T_3 record corrupted architecture state. If a fault is subsequently detected (through a division by zero anomaly detector), rolling back to T_3 would not result in recovery as the fault-free value of `b` is not restored. However, if the system rolls back to the checkpoint at T_2, the system would be recovered although the checkpoint records corrupted architecture state (the value of `a` is corrupted in this checkpoint) as the faulty value of `b` that lead to the detection was restored, assuming that `a` is not used for any other computation. Not accounting for these differences would make us draw unnecessarily conservative inferences on fault recovery as we would consider only the checkpoint at T_1 to be recoverable.

4.2.1 Soft-Latency: A Recovery-Oriented Definition of Detection Latency

In order to account for these differences, we define a notion of software state corruption to study fault recovery. We do not give a constructive definition of software state as the precise set of values that define the state of the software is software-specific and hard to determine. We simply define software state corruption as the point where a checkpoint would not result in recovery.

Figure 4.6 shows the time line for the execution shown in Figure 4.5, elucidating the differences

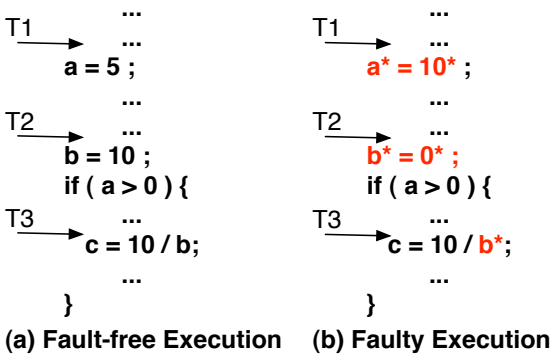


Figure 4.5: Example execution to show that the system may be recovered even from checkpoints that record corrupted architecture state. Although the checkpoint taken at T_2 records corrupted architecture state, the system would be recovered by rolling back to that checkpoint. Ignoring this property of recovery may lead to making unnecessarily conservative conclusions about the recoverability of detected faults.

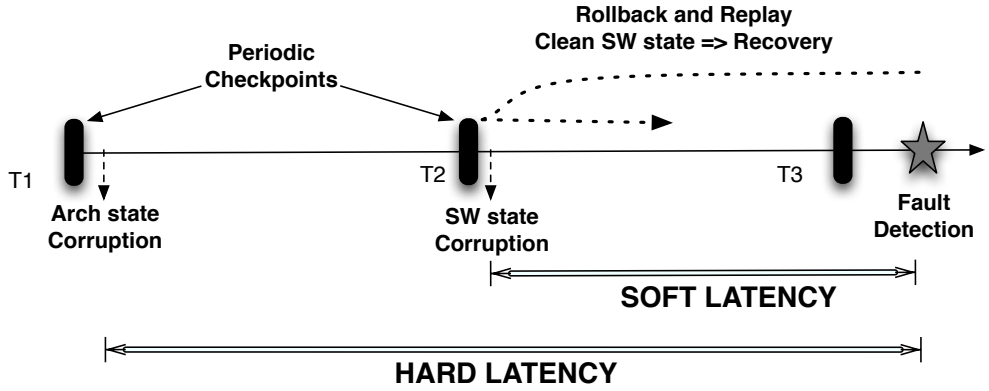


Figure 4.6: Difference between Hard- and Soft-Latency. Although the checkpoint at T_2 records corrupted architecture state, the software state is clean, resulting in successful fault recovery. Thus, the Soft-Latency is more relevant for studying fault recovery than the Hard-Latency.

between the Hard- and the Soft-latency. Since the architecture state is corrupted after T_1, the checkpoint taken at T_1 records clean architecture state. The *Hard-Latency* is the latency from when the architecture state is corrupted to when the fault is detected. Since rolling back to the checkpoint at T_2 would recover the software, we say that T_2 has recorded clean software state. The latency from when the software state is corrupted until when the fault is detected is the *Soft-Latency*. Rolling back to a checkpoint taken after the software state is corrupted, such as the one at T_3, would not result in successful recovery, making the software state recorded in this checkpoint tainted.

This latency between software state corruption and fault detection, or the *Soft-Latency*, is the

latency pertinent to recovering this fault. If the *Soft-Latency* is considerably lower than the *Hard-Latency*, the recovery intervals will be much shorter, resulting in lower overheads during fault-free execution.

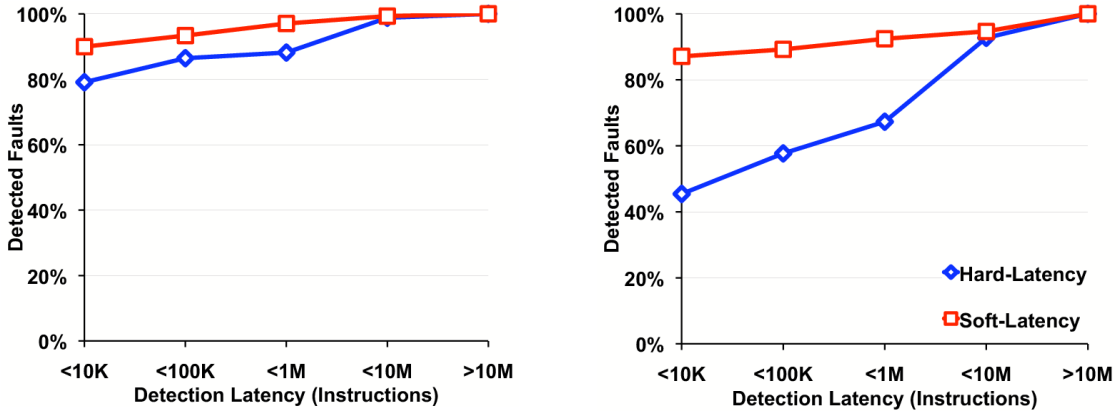
4.2.2 Evaluating the Soft-Latency

Measuring the Soft-Latency is, however, non-trivial as the state of the software is software-specific. Note, however, that this measurement is required only for *analyzing* the recovery interval and is *not required* in a real system. A real system would simply roll back by the identified recovery interval to recover from the detected fault.

In this work, we measure this latency *post facto*. We take periodic checkpoints of the entire system; once a fault is detected, we rollback to successive checkpoints and replay the full application without the fault for each such checkpoint. The most recent checkpoint for which this replay produces fault-free output provides the new latency. Application state is clearly not corrupted at that checkpoint and hence the soft-latency must be less than the interval between this checkpoint and detection. Note that this elaborate procedure is only required to measure the new-latency to guide recovery overhead analysis – a real system does not need to measure this detection latency. In the future, we can also leverage prior work on identifying the minimal amount of state required for correct application execution [10, 25, 36] for a more efficient measurement.

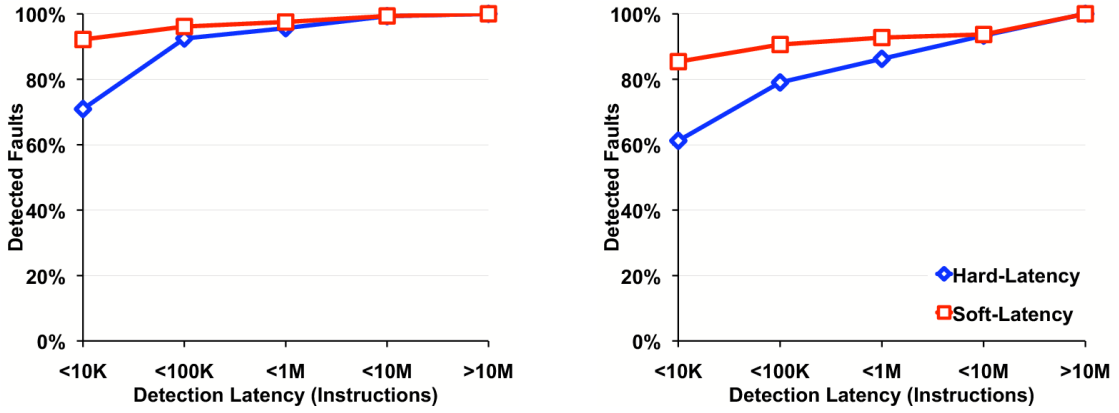
Figure 3.5 shows the distinction between the Hard-Latency and Soft-Latency for permanent and transient faults in (a) server and (b) SPEC workloads. In each graph, *Hard-Latency* shows the detection latency from architecture state corruption, and *Soft-Latency SWAT* shows latency from software state corruption.

From the figure we see that for faults detected in both the server workloads and the SPEC benchmarks, there is a significant difference between the Soft-Latency and the Hard-Latency – at any given latency, more faults are recoverable than previously reported, as shown by the difference between the Soft- and Hard-Latency curves. For example, for the transient faults detected in server workloads (right-side graph in Figure 3.5(a)), only 45% of detected faults have *Hard-Latency* of under 10K instructions, while nearly twice as many faults (87% of the detected faults) have *Soft-Latency*



Permanent Faults Transient Faults

(a) Server Workloads



Permanent Faults Transient Faults

(b) SPEC workloads.

Figure 4.7: Detection latency for (a) server and (b) SPEC workloads. With the new definition of detection latency, 90% of the faults are detected within 100K instructions for all workloads across permanent and transient faults. In contrast, 90% recoverability required millions of instructions for transient faults in SWAT. This latency reduction reduces recovery overheads significantly.

(and are hence recoverable with full client and server checkpointing) within 10K instructions. The graphs show such trends for nearly all other latencies in both the SPEC and the server workloads.

Additionally, these results show that over 90% of the faults have a Soft-Latency of under 100K instructions, making them recoverable with a full system checkpoint taken at this checkpoint interval. In contrast, with the old definition of latency (the Hard-Latency curves in Figure 4.7), 90% recoverability required millions of instructions, especially for transient faults. Thus, the overheads from fault recovery are actually lower than that predicted by the Hard-Latency, making this distinction important to consider.

These results show that for system recovery, it is not necessary to recover from *all* corruptions in the architecture state; the system may be recovered even without restoring some corruptions that may not affect software state. Ignoring this distinction and deciding the recovery interval based on the Hard-Latency may result in choosing unnecessarily conservative intervals for checkpointing, resulting in high overheads during fault-free execution.

4.3 Summary and Implications

In this chapter, we identify properties that enable certain applications to tolerate faults in data values and outputs. We apply these properties to fault detection and revisit the SDC rates and detection latency results presented in the previous chapter. Our results and analysis show that taking application-level fault tolerance is important for SWAT as it helps to identify the real SDCs from the previously reported potential SDCs. It also helps to accurately guage the overheads from fault recovery by making a distinction between architecture-state and software-state corruptions. Overall, the results show that SWAT achieves a low SDC rate of 0.2% (56/35,760 faults) for permanent and transient faults in SPEC and server workloads for faults in non-FPU structures, and demonstrates promise for high recoverability with short checkpoint intervals. This translates to a two orders of magnitude reduction in the FIT rate when compared to the base system with no protection against faults.[1] Further, several of these 56 faults may be identified with additional support

[1]Of the 17,880 permanent and 17,880 transient faults injected into the workloads, 2,571 permanent and 15,195 transient faults were masked. The remaining 17.521 faults are activated and affect software execution. We assume that they result in failures in the base system that is not protected by SWAT. With SWAT, however, only the 56

from the software, but we leave this exploration to future work.

The results presented in this chapter demonstrate the importance of taking application properties into consideration while designing future resilient systems. With future systems expected to breach existing boundaries of hardware and software abstractions to achieve solutions that incur low cost, leveraging application properties for fault resiliency is expected to become the norm. This chapter also paves the way for future research that leverages support from the application to achieve higher reliability targets at the cost of higher overheads.

SDCs are not handled and cause system failures. SWAT thus reduces The FIT rate by over two orders of magnitude.

Chapter 5

Fault Recovery

The goal of the recovery module is to restore the system to a fault-free state after a fault has been detected. Fault recovery is therefore an important part of any fault tolerance solution. Consequently, fault recovery is an extensively studied topic, spanning decades of research in commercial and client systems.

There are several alternatives to implementing effective fault recovery. First, mechanisms for fault recovery can be implemented either in software or in hardware. Software-level checkpointing schemes [11, 32, 41] are typically used for low frequency checkpointing (every few seconds or minutes) as they require a globally synchronous checkpoint. Hardware checkpoint is typically used for high frequency checkpointing (every few milliseconds, or lower) as they incur lower overheads [16, 51, 60, 71]. Another alternative is to implement employ either forward-error recovery techniques where the detectors provide sufficient information to recover the system instantaneously, or backward-error recovery where the fault is allowed to corrupt the state of the system system, but there exists support for checkpointing and rollback once a fault is detected.

While there are several constraints involved in the deciding between the various alternatives for fault recovery, perhaps the most important consideration is the overheads incurred from the operations that enable recovery. Since these operations that enable fault recovery are performed during the common-mode of fault-free execution, the recovery module should be designed such that it incurs minimal overheads in this mode. Additionally, the overheads from these operations are intrinsically linked to the fault detection scheme. On the one hand, while shorter detection latencies ensure that these overheads are lowered, the overheads from the detectors that are required to detect the faults at shorter latencies may become too high. On the other hand, longer detection latencies may lower the overheads from the detectors, but recovery may require restoring much of the system

state, making the overheads from fault recovery prohibitive. There is thus an intricate relationship between fault detection and recovery, making it important to consider the detection scheme when designing and evaluating a scheme for fault recovery. Much prior work has ignored this relationship, making their evaluations incomplete.

SWAT is designed a complete solution that encompasses detection, recovery, and diagnosis. We thus design fault recovery in light of fault detection, taking design decisions that consider the relationship between them. SWAT employs low-cost detectors that allow the fault to corrupt the state of the system and identify the fault be observing anomalous software execution. Since the goal of SWAT is to achieve low-cost for overall system resiliency and the detectors detect faults at sufficiently short latencies (Chapter 3), SWAT relies on hardware support for fault recovery. Further, since the SWAT detectors allow the fault to corrupt the state of the system may not provide sufficient information to the recovery module for instantaneous recovery, SWAT uses BER schemes that rely on support for checkpointing and output buffering for fault recovery.

In this chapter, we take a closer look at the various components in SWAT's recovery module (Section 5.1). Of particular importance is the hardware support that SWAT uses for output buffering. Although output buffering is essential for fault recovery, much prior work has not ignored output buffering. We also evaluate the overheads incurred from the operations for fault recovery during fault-free execution (Section 5.3.1). Finally, we tie the fault recovery and detection together and present SWAT as a solution for in-core fault resiliency capable of detecting and recovering from permanent and transient faults even in the presence of I/O (Section 5.3.2).

5.1 SWAT Recovery Components

The SWAT recovery module contains 4 key components - processor checkpointing, memory checkpointing, device recovery, and output buffering. Once a fault is detected, each component of the recovery subsystem must take appropriate action to restore the state of the system to its checkpointed state – the processor and memory are restored from their checkpoints, devices are reinitialized, and buffered outputs are discarded. We refer to the interval by which the system is rolled back as the *recovery interval*. Note that the recovery interval may span multiple checkpoint intervals. Fur-

Component	Fault-free Operation	Recovery Operation	Fault-free Overheads	
			Performance	Area
Processor	Register snapshot	Restore registers	Pipeline flush	Negligible
Memory	Record undo log for stores	Replay undo log	[60, 71]	[60, 71]
System Devices	Monitor open connections	Reset devices, restore connections	Negligible	Negligible
Outputs	Buffer outputs	Discard unverified outputs	Outputs delayed	Impl. dependent

Table 5.1: Components of fault recovery. SWAT relies on hardware support for checkpointing and for output buffering to achieve low cost fault recovery.

ther, since all components are on during the common mode of fault-free operation, their overheads, dictated directly by the recovery interval, must be minimized.

Table 5.1 gives a high level overview of each component of SWAT's recovery module, along with its overheads. The following sections describe each component in more detail. We rely on existing work for recovering the processor, memory and devices, and propose a new technique for buffering externally visible outputs in hardware.

5.1.1 Processor Checkpointing

A processor checkpoint periodically records all the architecturally visible registers of a processor. Since the pipeline is flushed at each checkpoint, checkpointing incurs a small performance overhead. The area overheads from the recorded registers state is fixed per checkpoint and is rather small. Further, since processor checkpointing may be used for other optimizations (such as run-ahead execution [49]), its cost is amortized.

5.1.2 Memory Logging

A memory log is an undo log that records changes to memory since the last checkpoint. The log records the old value at a memory location that the first store to that memory location since the last checkpoint is about to modify. The overheads from memory logging are dependent on how logging is implemented. Hardware-based implementations like SafetyNet [71] incur negligible performance overheads. In-memory schemes like ReVive [60] require flushing all dirty cache states to the memory at the checkpoint, incurring higher performance overheads especially at shorter checkpoint intervals. SWAT's memory logs are implemented like the logs in SafetyNet, incurring near-zero performance overheads for short checkpoint intervals but some area overheads to store the logs.

5.1.3 Device Recovery

Once a fault is detected, the devices in the system need to be restored to a consistent state. Checkpointing devices by taking a periodic snapshot of device registers may be expensive as it would require modifying a large number of hardware devices. As a practical alternative, the devices can simply be reset during fault recovery and the associated driver can be recovered to a state consistent with the reset device by using software, as demonstrated by prior work [30, 51, 74]. These scheme relies on higher levels of the software stack to replay inputs lost during recovery and to handle duplicate outputs during replay. Our recovery strategy is to rely on such software support to reset and recover the devices. Since this software that orchestrates the device recovery is recovered from the processor checkpoint and the memory logs, we can leverage this solution even in the presence of faults.

5.1.4 Output Buffering

An important, but commonly ignored aspect of fault recovery is output buffering. Buffering external outputs until they are known to be fault-free is essential to avoid the *output commit problem* which states that committed external outputs (such as a network packet, or a display on the screen) cannot be rolled-back during recovery. Buffering for the duration of the recovery intervals guarantees that these outputs are fault-free when they are released to the external world and would not require undoing. Since device outputs (that potentially affect waiting clients) are delayed, the performance overhead incurred from output buffering is directly related to the recovery interval.

To our knowledge, Revive-I/O [51] is the only existing work in a modern system that relies on hardware checkpointing that handles output buffering with relatively low performance impact. Revive-I/O buffers outputs in software, using a pseudo-device driver (PDD) to intercept output calls to device drivers which it holds till the end of the recovery interval. At that time, the PDD calls the actual device driver to process the intercepted output call and release the outputs. Revive-I/O makes a subtle, previously unexplored, assumption that this device driver software will run fault-free. Unfortunately, the device driver runs on a potentially faulty core and is vulnerable to generating corrupted outputs – once these corrupted outputs become user visible, there is no way to

recover them. This vulnerability can be potentially serious as the device driver is a complex piece of software, and maybe affected by hardware faults as much as any other code.

Buffering Outputs in Hardware

SWAT uses an alternate solution to the output commit problem that does not have the above vulnerability. In contrast to the high-level software buffering, it buffers low-level output requests (stores to I/O space) in hardware until the end of the recovery interval. The hardware based solution has the advantage that it is device-oblivious and is hence easily extendable to a variety of devices; in contrast, software-level solutions deal with high-level device operations and must be aware of the semantics of these operations. Further, the hardware solution is arguably simpler than the software-level solution as it does not need any complex changes to device software. While we still need to protect the hardware buffer and the (very simple) finite state machine controlling the buffer, this is far easier than protecting the full core running the complex software device driver.

Modern processors communicate with external devices (such as the ethernet card, disk, etc.) through accesses to dedicated memory locations that are uncacheable. Inputs from the devices are read using load instructions, and outputs are issued through store instructions. The hardware buffer thus buffers stores from the CPU to these dedicated memory locations to handle the output commit problem. This buffer is implemented in the core and requires no changes to device hardware, as shown in Figure 5.1(a). It thus supports output buffering for any I/O device that sits on the PCI bus with no modifications. (Certain user-level devices that are not implemented on the PCI bus, but are implemented on the memory bus may not be amenable to this form of output buffering.) The fault-free and recovery operations of this hardware output buffer are described below.

Fault-free Operation: During fault-free execution, the buffer delays *all* stores from the CPU to the devices for the duration of the recovery interval. Subsequent reads to addresses with buffered stores return the values from the buffer. The buffer may be drained under two conditions. First, at the next checkpoint interval, the buffered stores are verified to be fault-free and would not require rolling back. Second, the ISA may require that at certain memory fences, all previous I/O instructions should be completed in order to guarantee that their side-effects have occurred (as some

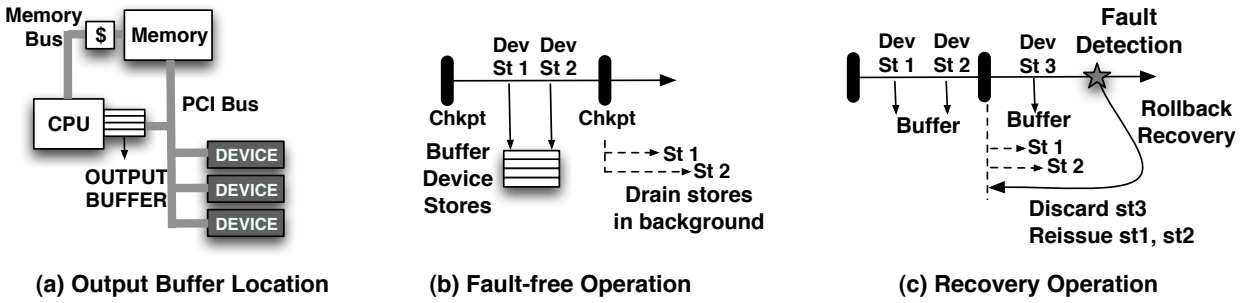


Figure 5.1: A hardware-level implementation of output buffering. The proposed buffer is a simple hardware structure that delays device outputs until they are verified to be fault-free.

I/O stores may implicitly affect the contents at other addresses). Under either of these conditions, the outputs are released to the devices in the background (much like how the store buffer for data memory in a modern processor drains its stores) so that the execution is not stalled to drain the outputs, as shown in Figure 5.1(b).[1] Although some of these stores may not result in externally visible outputs and may only configure the devices, the hardware cannot identify such outputs owing to the lack of semantic information about the outputs.

Once the buffered store is issued to the device, the device may access certain resources (such as in-memory buffers) that need to be transferred to the device. Since this transfer is not atomic, the hardware fault in the core may corrupt the resource during the transfer and make the fault externally visible. The system should therefore be designed such that the outputs are protected during the period of this transfer to avoid the output commit problem. Further, this protection would have to be implemented with minimal hardware without trusting the software as it may be executing erroneously under the influence of the hardware fault.

The only such instance in our system that warrants special attention are DMA transfers from the CPU to the devices. DMA transfers are initiated by the CPU setting up an in-memory buffer with outputs and then allowing the DMA engine or the device to transfer the buffer to the device (during which time the PCI bus is controlled by the DMA engine or the device). Although the in-memory buffer is locked with page-level protection mechanisms that prevent any CPU stores from modifying the buffer [13, 14], the hardware fault may corrupt the protection bits used to protect the

[1] I/O fences may artificially shorten the checkpoint interval and affect recoverability. However, such fences account for $< 0.01\%$ of our dynamic instructions in our workloads, making this loss insignificant.

DMA buffer. Consequently, CPU stores may erroneously modify the contents of the DMA buffer before it is transferred to the device, making the fault externally visible.

We leverage the synchronization mechanism employed between the CPU and the device to implement a low-cost solution that avoids such corruptions. A simple hardware that ensures that the location of the protection bits is not modified during this transfer would suffice to prevent such a corruption. (The location of the protection bits can be identified by monitoring the page table when the DMA buffer is allocated through dedicated system calls.) Since this hardware cannot trust the software to identify when these checks should be turned on and turned off, it monitors the store that modifies the protection bit to turn on the checking and turns off the monitoring when the interrupt from the DMA engine to acknowledge the completion of the transfer is received.[2]

Recovery Operation: Once a fault is detected, the outputs buffered in the current recovery interval are discarded to prevent committing faulty outputs. The processor and memory are then rolled back to a pristine checkpoint, as shown in Figure 5.1(c). During this rollback, outputs committed at the end of the previous recovery interval are nullified. They are thus reissued to bring the system to a consistent state; committed stores are therefore retained for one extra recovery interval in the hardware buffer. The system then invokes existing software-level solutions to recover the devices through a device reset and by reissuing any on-going logical operation [51, 74]. Similar to these solutions, we rely on higher level protocols to handle duplicate outputs and replay inputs.

Fault Tolerance: The hardware buffer is thus a simple piece of hardware that can be periodically tested for faults with previous techniques that identify faults in control structures [12]. Further, outputs in the hardware buffer are ECC checked to protect from faults in the buffer. The committed outputs are thus protected from faults in the processor, making our technique attractive.

Overheads: Although a hardware output buffering solution appears like a conceptually simple implementation of software-level output buffering in hardware, its feasibility for software anomaly detection techniques is far from clear. Software-level buffering, on the one hand, delays each high-level device operations (such as a single disk write, or sending a packet using the ethernet device)

[2] Although a device-level fault may result in faking this interrupt, the focus here is on faults within the core. Faults in off-core components require further exploration that is beyond the scope of this thesis.

by one recovery interval. Since each such high-level output may correspond to multiple low-level stores (one high-level disk write, for example, may correspond to multiple device stores to configure the disk controller, etc.), hardware buffering may incur delays of multiple recovery intervals for each high-level output, imparting potentially high delays to waiting clients. Further, buffering outputs in hardware may require large buffers (as multiple stores may be delayed in each interval), incurring high area overheads; the buffer size is less important in software-level solutions which store the buffers in main memory. The performance and the area overheads need to be evaluated and minimized so that fault-free execution is not affected.

5.2 Evaluating SWAT Recovery

5.2.1 Simulation Environment

We implemented the SWAT recovery module within a simulation framework that models a full system, including the CPU, the memory, and peripheral I/O devices. We used the simics full system simulation framework to model a distributed system with a server and client system communicating through a simulated ethernet link with a latency of 0.1ms. The server and client systems have identical configurations with UltraSPARC-III-like processors, 4GB of physical memory and I/O peripherals.

In this configuration, we study 4 I/O intensive server workloads with multithreaded drivers running on the client and transfer 10s of MBs of data from the server to the client. We restrict our evaluation of fault recovery to these I/O intensive workloads as their recovery is known to be hard due to the presence of I/O. Table 3.2 gives a description of the operations of our workloads and Table 5.2 gives more details about their data-transfer rates. Although all the client drivers, except that for mysql, are multithreaded, the client systems spend a significant fraction of time waiting for I/O from the server, much like today's systems. These measurements are made without any added delays from output buffering.

Real-world server workloads may, however, contain multiple server daemons or commercial database backends that may be dependent on each other and serve thousands of requesting clients

Application	Data Transfered	Average Rate	Percentage of execution waiting for I/O
apache	38MB	9.57MBps	76.53%
sshd	19MB	2.49MBps	24.33%
squid	20MB	11.64MBps	69.50%
mysql	7.5MB	1.05MBps	71.13%

Table 5.2: I/O statistics of our server workloads. Although the client drivers are all multithreaded (except for *mysql*), the client drivers spend a majority of their time waiting for I/O from the sever.

while transferring gigabytes of data. Our server workloads, however, contain only one daemon that processes client requests and do not model interactions between different daemons. We nevertheless verified that our workloads exhibit I/O trends similar to published results from commercial workloads, making them representative of real-world workloads. Figure 5.2, for example, shows the cumulative distribution of the inter-arrival rate of I/O operations (in instructions) for our workloads. These rates are similar to published rates of the inter-arrival rate of I/O operations in commercial TPC-C workloads for IBM-DB2 [70], demonstrating that our workloads exhibit similar trends in I/O operations when compared to commercial workloads. Further, even when performing measurements with workloads that transfer gigabytes of data, only a small fraction of the overall execution is studied owing to limitations of simulation speed. In these intervals, these real-world workloads would appear like the ones that we simulate within our infrastructure.

5.2.2 SWAT Recovery Implementation

Within this simulation framework, we implement fault recovery in the server system to model a scenario where the server that serves remote clients is protected from hardware faults by SWAT. We therefore model the server system in detail with the GEMS timing models for the processor and memory (Table 3.1).

Processor and Memory Checkpoint

For the processor checkpoint, we take a periodic snapshot of the processor's architecture registers. The memory logging is implemented with a SafetyNet-like logging scheme. The first stores to a

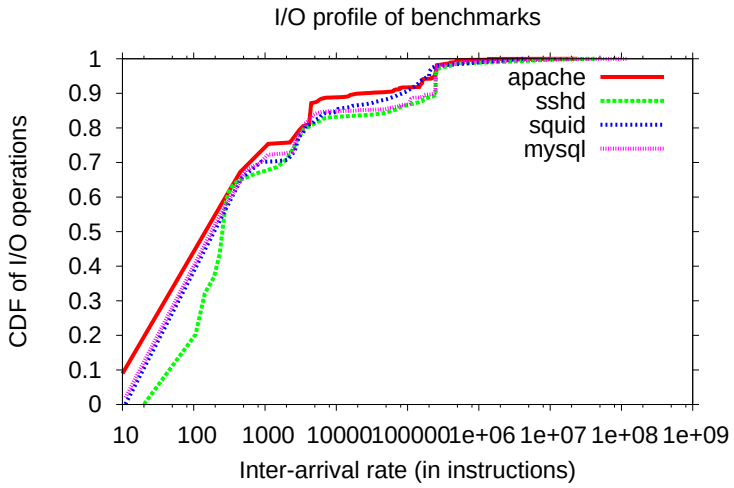


Figure 5.2: Inter-arrival rate of I/O operations for the server workloads. Although our workloads are of smaller scales than commercial workloads, they exhibit similar trends for I/O operations when compared to commercial workloads.

memory location since the last checkpoint triggers entering the old value at that memory location into a log. This enables undoing the store at the time of recovery. While this incurs near-zero performance overhead, the CLBs that store the memory logs may incur considerable hardware overhead if they are implemented entirely in hardware. This overhead may be reduced by maintaining a small hardware CLB that is periodically flushed it to a dedicated portion of the main memory. (Section 5.3.1 discusses this further.)

Hardware Buffering

The operations of the hardware output buffer is simulated with a new Simics module in the server that buffers writes from the processor to devices. At each checkpoint, writes buffered in the second-to-last checkpoint interval are committed assuming that the recovery interval spans less than two checkpoint intervals. Although the recovery interval can technically span arbitrarily many checkpoint intervals, the outputs would have to be buffered for the entire duration, motivating this choice for our evaluations.

In addition to draining at the checkpoint and at memory fences that enforce orderings between I/O operations, the hardware output buffer is drained when loads that attempt to read values

from buffered stores (as they pertain to the same address) are encountered. Simics limits us from performing this store-to-load forwarding accurately, forcing us to drain the output buffer when we encounter such dependencies. However, we see such loads that not preceded by memory fences to be rare, making our loss in recoverability insignificant.

Device Recovery

Due to infrastructure limitations we implement device recovery by taking a periodic snapshot of the device registers in the server and restoring them during rollback, instead of resetting the devices and reinitializing the drivers.

5.2.3 Metrics for Evaluation

Overheads during Fault-free Execution

We do not measure the overheads from periodic register checkpointing as it is known to incur near-zero performance and area overheads. The register checkpoints can be stored in dedicated memory locations that are ECC protected.

The overheads from hardware output buffering and memory logging on fault-free execution are measured by simulating the entire application without any faults. Since attaching timing models prohibitively slows down the simulation, the processor and memory are simulated functionally with each instruction and memory operations taking 1 cycle to complete. While more accurate models would yield more accurate results, we believe that the qualitative nature of our results would still hold.

The performance overhead from output buffering is measured as the ratio of the time taken by the client to receive all its requested files from a server which performs output buffering in hardware, when compared to requesting files from a server that does not buffer outputs. While the buffered stores would be drained in the background in a real system, they are drained instantaneously in our implementation; we conservatively charge one cycle to drain each such store. The area overhead of the hardware buffer is measured by recording the sizes of data and addresses buffered.

For the memory logs, we measure the log sizes from memory writes (from the CPU and from

DMA) for varying checkpoint intervals. We do not focus on the performance overheads from memory logging as prior work has extensively studied these overheads and has devised mechanisms for memory checkpointing that incur low overheads [71].

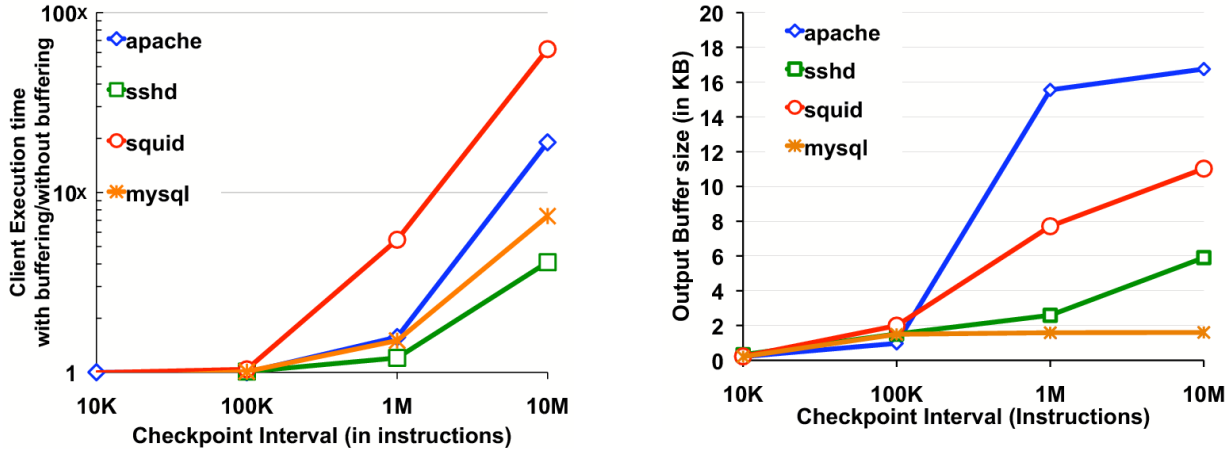
Recoverability

The SWAT detectors and the recovery module that buffers external outputs in hardware provide a solution to detect and recover from hardware faults. We evaluate this combined system by injecting 8,960 permanent and 8,960 transient hardware faults in the server workloads, with the infrastructure described in Chapter 3. Since we inject and detect faults only in the server, we recover only the server system without affecting the client.

The injected faults are detected using the SWAT detectors discussed in Chapter 3. For faults detected within the 10M instruction window of detailed simulation, the system is recovered with checkpoints taken at different intervals from 10K instructions up to 10M instructions.

Once a fault is detected, SWAT restores the processor, memory, and devices from their checkpoints, discards unverified outputs as they may be faulty, and recommits outputs buffered in the previous recovery interval. We then turn the fault off and simulate the system until either a software anomaly is triggered or the application completes. The fault is recoverable if the application output matches the fault-free output. Since the purpose of this functional simulation is to collect the output of the application, output buffering is turned off after the fault is detected. In addition to this *Full* system, we study two other systems to evaluate the importance of each recovery component – the *No-Device* system that buffers outputs but does not recover devices, and the *No-I/O* system that neither buffers outputs nor recovers devices (both systems recover the architecture state).

The faults are then classified into 4 categories based on their outcomes. Faults masked by the architecture or the application are classified as *Masked*. Detected faults that are successfully recovered at the given checkpoint interval are classified as *Detected + Recovered*, and those that are not as *Detected + Unrecovered*, also known as DUEs. (All faults detected at a latency of $> 10M$ instructions are trivially DUEs.) The remaining faults corrupt the output of the application and are classified as *Potential SDCs*. As discussed in Chapter 4, some of these faults may be tolerated



(a) Performance Overheads

(b) Area Overheads

Figure 5.3: Overheads from hardware output buffering on fault-free execution. The low performance overheads and log sizes for checkpointing intervals of under 100K instructions motivate designing systems with these checkpoint intervals.

by the server workloads through simple retries. We account for this application-level tolerance and measure the true SDC rates as well.

The recoverability of the SWAT detectors is measured as a fraction of the injected faults that are either masked or are detected and recovered by SWAT. Such faults are externally invisible, demonstrating the efficacy of SWAT to contain the fault within the server. Owing to the large number of injected faults, we see a statistical error of under 0.7% at a 95% confidence intervals for the recoverability, giving us high confidence in our results.

5.3 Results

5.3.1 Overheads During Fault-free Execution

Hardware Output Buffering

Figure 5.3 shows the overheads from buffering outputs in hardware for several checkpoint intervals. Figure 5.3(a) shows the performance overheads while Figure 5.3(b) shows the area overheads.

Server-side buffering incurs overheads as responses to clients are delayed. From Figure 5.3, we see that buffering external outputs in hardware incurs low performance and area overheads at

Workload	Maximum Size (in KB)	95th Percentile (in KB)
apache	121	21
sshd	141	56
squid	170	47
mysql	152	16

Table 5.3: 95^{th} percentile of memory log sizes. The area overheads from storing the memory logs in hardware may be reduced by keeping a small hardware buffer that is periodically flushed to a dedicated portion of the memory.

checkpoint intervals of under millions of instructions (<5% performance and $< 2KB$ area overheads for intervals of 10K and 100K). However, at longer intervals of millions of instructions, the overheads incurred are much higher. At an interval of 1M instructions, the performance overheads are as high as 5X and rise up to 62X at a 10M instructions checkpoint interval. The area overheads plateaus at a maximum of 18KB for intervals of millions of instructions.

These overheads from buffering outputs in hardware may in fact be higher than those incurred from software buffering. To understand the reasons behind these overheads, consider a checkpoint interval of C and a high-level output command comprising several low-level stores. Assume some of these stores depend on a client response acknowledging a previous store (e.g., like TCP flow control), creating a dependence chain. Software buffers the entire high-level command, delaying the entire dependence chain by a total of 2 checkpoint intervals or 2×C. Hardware buffering will delay each server-store to client-response dependence by two checkpoint intervals, resulting in additive delays of 2×C×L for a dependence chain of length L. Hardware buffering is nevertheless more attractive as buffering in software assumes that the software draining the outputs is fault-free.

Memory Logging

Figure 5.4 and Table 5.3 show the sizes of the memory logs during fault-free execution. Figure 5.4 shows the maximum log sizes, while Table 5.3 lists the maximum and 95th percentile of the log sizes for a checkpoint interval of 100K instructions.

From Figure 5.4, we see that checkpoint intervals of under 100K instructions give a maximum log size of 170KB. Increasing this interval to millions of instructions, however, results in log sizes of several megabytes For example, the maximum log size at a checkpoint interval of 10M instructions

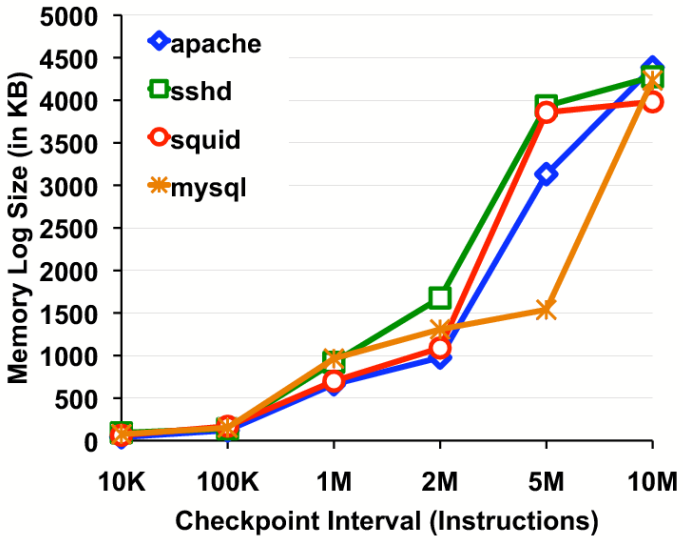
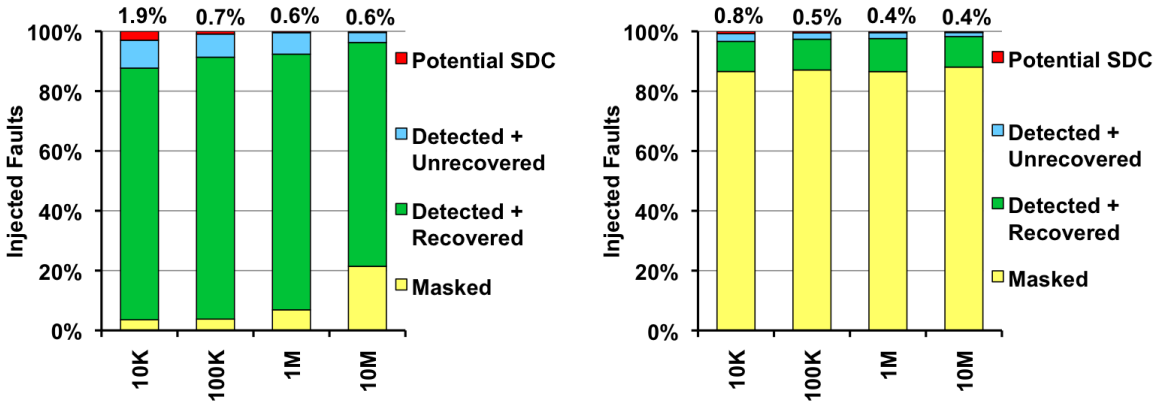


Figure 5.4: Maximum Sizes of the memory logs for various workloads (in KB). The memory logs grow to MegaBytes in size for checkpoint intervals of millions of instructions.

is 4.4MB.

The true overheads from memory logging, however, depend on how the logs are implemented. SafetyNet proposed that the CLBs that store the memory logs be completely implemented in hardware [71]. The resulting overhead of 170KB per checkpoint interval may be too high. This overhead can be further reduced by using a small hardware buffer that is periodically flushed to memory. Table 5.3 shows the 95th percentile for memory logs for a checkpoint interval of 100K instructions. (The maximum sizes are also shown for comparison.) From this table we see that sizing this hardware buffer at 57KB, which is the maximum 95th percentile for the memory logs for our server workloads at a checkpoint interval of 100K instructions, would require infrequent flushes of this buffer to memory (for $< 5\%$ of the checkpoint intervals). The area overheads from the memory logs is therefore significantly reduced with minimal impact to fault-free execution.

These results show that the checkpoint interval for any practically acceptable hardware recovery solution should be under millions of instructions so that fault-free execution is minimally affected. Previous work assumed, however, that checkpoint intervals of millions of instructions may be acceptable as they ignored the impact of output buffering on fault-free execution.



(a) Permanents (b) Transients

Figure 5.5: Outcome of detecting and recovering injected permanent and transient faults. At a checkpoint interval of 100K instructions, SWAT detects and recovers over 94% of the injected faults with $< 0.2\%$ of the faults resulting in unacceptable SDCs.

5.3.2 Recoverability of SWAT

Overall Results

The bars in Figure 5.5 shows the efficacy of SWAT to detect and recover the system from in-core permanent (Figure 5.5(a)) and transient (Figure 5.5(b)) faults in the presence of I/O. The numbers on top of this bar for each checkpoint interval is the fraction of injected faults that either escape detection and result in potential SDCs, or are detected but the output of the recovered application differs from the fault-free output.

The graphs show that the the fraction of injected faults that result in potential SDCs decreases as we increase the checkpoint interval from 10K instructions till 10M instructions. The higher masking rate for permanent faults at the 10M checkpoint interval is an artifact of our simulation window of 10M instructions coinciding with the checkpoint interval. Consequently, buffered outputs are not released within our simulation window, limiting the amount of forward progress that the applications make at this checkpoint interval.

The overheads results from Section 5.3.1 showed that checkpoint intervals of 100K instructions or lower is preferable to minimize the impact on fault-free performance. Focusing on a checkpoint interval of 100K instructions (motivated by Section 5.3.1), we see that the detection and recovery

solutions are effective for hardware faults. Even at a this low checkpoint interval, 94% of the injected faults are either masked or are detected and recovered by SWAT without affecting application output. Although 5% of the injected faults are DUEs (Detected Unrecovered Errors) that may have affected outputs prior to detection, SWAT can inform the clients to discard the outputs as they may be faulty. Of the remaining faults, only 44 faults (0.2% of the injected 17,920 faults) are true SDCs. Shorter checkpoint intervals yield larger potential SDC rates (1.2% at a checkpoint interval of 10K instructions from Figure 5.5), making the 100K instruction interval the sweet-spot for checkpointing.

Our system therefore achieves high recoverability rate, and low SDC rate for both permanent and transient faults while incurring low overheads during fault-free execution.

Importance Of I/O For Fault Recovery

Figure 5.6 shows the importance of device recovery and output buffering for system recovery in the presence of I/O. For each checkpoint interval, the bars show the outcomes from detecting and recovering permanent faults injected into *No I/O* system that recovers only the architecture state without buffering outputs or recovering devices, the *No Dev* system that recovers the architecture state and buffer outputs but does not recover devices, and the *Full* system that recovers the architecture state and devices, and buffers outputs.

From the figure, we see that buffering external outputs and device recovery are essential for system recovery in the presence of I/O across all checkpoint intervals. Removing device recovery reduces the fraction of recovered faults by as much as 89% as seen in the *No-Device* system. In the absence of output buffering and device recovery, i.e., in the *No-I/O* system, the fraction of unrecovered faults is further reduced by as much a 37%. These components of fault recovery have, however, been ignored by much prior work as they did not study recovery in the presence of I/O. Further, no prior work has evaluated the extent to which each of these components are required for fault recovery as demonstrated here.

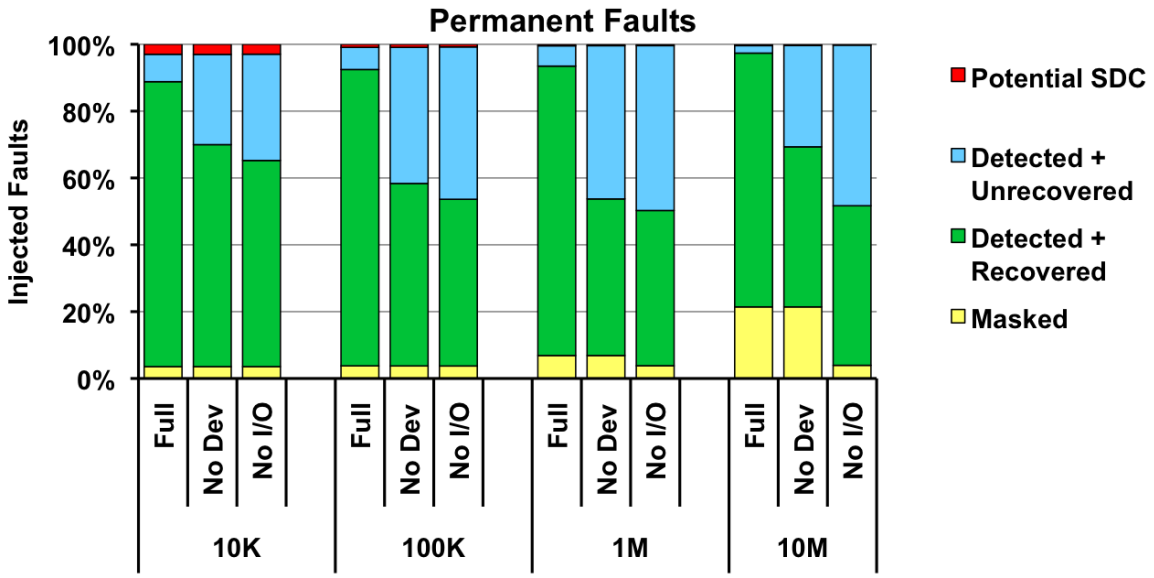


Figure 5.6: Importance of device recovery and output buffering for system recovery in the presence of I/O. The No-Device and No-I/O systems show that device recovery and output buffering are required for system recovery and cannot be ignored.

5.4 Summary and Implications

SWAT relies on BER schemes for fault recovery as the detectors allow the fault to corrupt system state prior to detection. SWAT therefore requires support for checkpointing and output buffering to recover the system once a fault is detected. This recovery module of SWAT is intricately related to the fault detection scheme as the checkpoint intervals for fault recovery, and consequently the overheads on fault-free execution, are dictated by the latencies at which the faults are detected.

This chapter presented the various components involved in SWAT's recovery module. Of particular emphasis is the hardware module used for output buffering that does not suffer from the identified limitations of existing software-level techniques. The simple buffer reduces the vulnerability of external outputs to faults and incurs low performance and area overheads at checkpoint intervals of under millions of instructions. We then present a comprehensive solution for in-core fault resiliency by evaluating the fault detection and recovery modules together. Our results show that the SWAT system achieves high recoverability, incurs low overheads during fault-free execution, and results in low SDC rates for in-core permanent and transient hardware faults.

Although the recovery results are presented in the context of SWAT, we believe that the principles of fault recovery presented here are applicable to other software anomaly detection and recovery schemes as well. We demonstrate how output buffering may be done with low-cost hardware, reducing the vulnerability of externally visible outputs. Further, this chapter shows that fault recovery and detection are two aspects of system reliability that go hand-in-hand and and must be studied together.

Chapter 6

Understanding When and Where SWAT Works

The SWAT system has so far been demonstrated to be large successful in detecting and recovering from permanent and transient faults for a variety of single-threaded and multi-threaded workloads running on single and multi-core systems. (Although this thesis does not deal with faults in multi-core systems interested readers are referred to the mSWAT paper that demonstrates SWAT's effectiveness in such scenarios [28].) We saw in the previous chapters that the SWAT detectors achieve SDC rates of under 0.5% for permanent and transient faults in a variety of workloads.

Despite the promise of low SDC rates at low overheads, the evidence for the success of SWAT and other software anomaly detectors has been through statistical fault injection experiments and are therefore, largely empirical. An intuitive understanding of the hard-to-detect faults that result in SDCs has not been built this far and is important for software anomaly detection for the following reasons. First, the SDC rates, although low, may not be low-enough for certain application classes such as mission critical systems, and finance applications. Understanding these SDCs may yield better detectors that lower the SDC rates further. Second, this understanding can help select the instructions that need to be protected through software-level redundancy methods [24, 44, 61, 66] in order to make the application more resilient to hardware faults. Third, the statistical fault injectors used to evaluate the detectors can be guided to inject these hard-to-detect faults and stress-test the detectors.

There are two axes along which the characteristics of these SDCs need to be studied – a hardware-centric axes and an application-centric axes. A hardware-centric characterization would help to understand how hardware faults manifest themselves as faults in the application by studying which instructions are affected. An application-centric characterization would then help to understand how faults at the instruction level manifest themselves as visible software anomalies that may be

detected. These hardware-centric classification would help to identify those hardware structures in which faults may be harder to detect and can be used to choose which hardware structures to harden against faults. The application-centric classification, on the other hand, would help identify application values in which faults may be harder to detect, enabling application hardening against hardware faults.

In this chapter, we build such an intuitive understanding of faults and answer the following questions – Why are software anomaly detectors, like SWAT, effective in detecting hardware faults? Where do they fall short? What values should we target for future detectors? Are there hardware properties that may be exploited for improving fault resiliency? We first take an application-centric view of faults and understand how faults propagate through the application and result in detectable anomalies (Section 6.1). The key insight used is that software anomaly detectors, like SWAT, rely on some form of deviation in control flow or memory addresses for successful detection. Faults in values that do not affect such operations, classified as *data-only values*, are therefore vulnerable under software anomaly detection. We then measure the frequency of these data-only values in real workloads, and study how faults in such values may be detected (Section 6.2). We then take a hardware-centric view of faults and demonstrate that faults in certain structures that are sparingly utilized (because of microarchitecture-level redundancies) may result in higher SDC rates (Section 6.3). Enforcing a uniform utilization of such structures may help reduce the SDC rates, making software anomaly detection more effective.

6.1 An Application-Centric View of Faults

In this section, we take an application-centric view of faults to how SWAT detects faults that corrupt the application. For this purpose, we first assume that the hardware fault has resulted in a fault in the application instruction. We present a model of how such a fault in the instruction word may affect the execution of software in order to build some intuition behind the propagation of faults through the application. Subsequently, we identify some limitations in the existing SWAT detectors based on this model and study strategies to overcome the identified limitations.

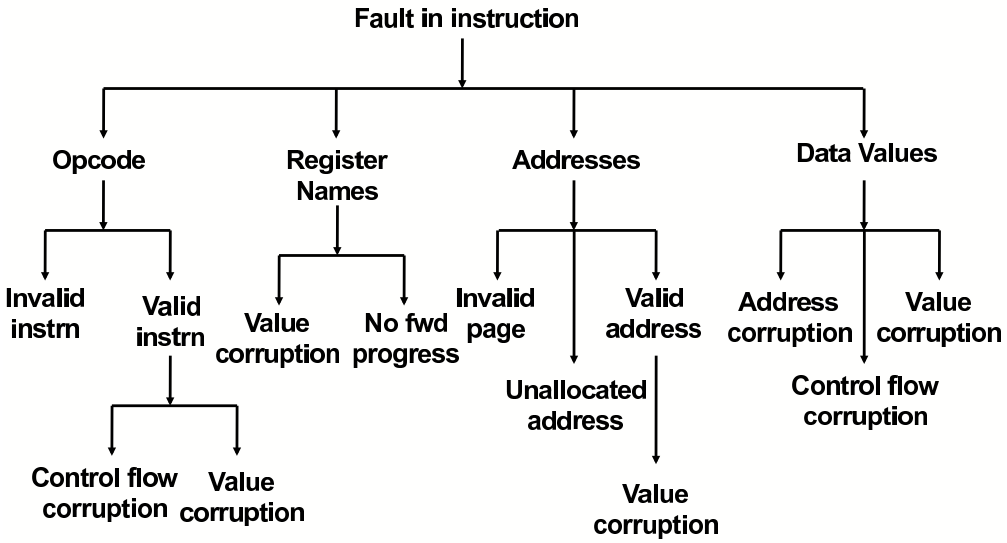


Figure 6.1: A fault may corrupt opcode, register names, address, or data values (of src/dest registers). Each such corruption propagates through the application in a variety of ways. The SWAT detectors identify corruptions in most values but are limited in their ability to identify pure data value corruptions.

6.1.1 A Model of How Faults Propagate Through the Application

Any instruction consists of four parts – opcode, register names (sources and destinations), address computed for memory operations, and data values of source and destination registers. The following sections describe the impact of corrupting each of these parts. Figure 6.1 gives a high level overview of the propagation.

Corruption in Opcode

A fault that affects the opcode can either mutate the instruction to an invalid instruction, that results in an *Illegal Instruction* fatal trap, or to another valid instruction. Valid mutations may result in either value corruptions, if the mutation involves non-branch opcodes, or in control flow corruptions if the mutation involves branches. Invalid control flow may result in *Illegal Instruction* or *App Abort* anomalies. Other types of corruptions are analogous to data value corruptions discussed below.

Corruption in Register Names

A fault may change the name of an architecture register used or produced by an instruction (e.g., source register changes from r_5 to r_9). If a different register is read, then the effect of the fault is similar to a data value corruption in the destination register. However, if a different register is written, the fault corrupts the data values in two registers. All such corruptions propagate as faults in data values.

In architectures with register renaming, the hardware fault affects the renamed register numbers (of either source or destination registers) resulting in instructions indefinitely waiting for either free registers (if the source register number is free) or in data value corruptions (if the destination register number is currently in use). The former instances of lack of forward progress can be identified with a *Watchdog Timer*, while the latter results in data value corruptions.

Corruption in Address

For a memory instruction, the fault may mutate the address into an address on an invalid page, on a valid unmapped page, or a valid mapped page.

In architectures like SPARC, accesses to misaligned addresses are treated as illegal accesses, resulting in a *Misaligned address* trap for such mutations. Further, if the address translation misses in the TLB, the software managed TLB invokes the OS to handle the miss. Since the OS uses the same faulty hardware, it could further activate the fault and result in anomalous software execution. Several detectors in SWAT detect such anomalous execution from the OS – *Kernel Panic*, *High OS*, *System Hang*, and *RED State*.

Invalid Page: Invalid pages are identified through a page table walk by either the hardware or software, depending on TLB management. Hardware managed TLBs would give a trap indicating an invalid page on such addresses. Software managed TLBs invoke the OS for this identification that may either activate the fault and result in anomalous software execution, or may identify the illegal address and abort the application – identified by the *App-Abort* detector.

Valid Unmapped Page: Although the corrupted address may be in a valid page, the page may

currently not be mapped in the page table, incurring a page fault. The OS is then invoked to handle such page faults. Since the OS has elaborate control flows, the hardware fault is further activated by the OS result in several anomalous executions from the OS which are detected through the *Kernel Panic*, *High OS*, *System Hang*, and *RED State* detectors.

Valid Mapped Page: If the address is (allocated or unallocated) on a valid page that is currently mapped, the effect of the fault is analogous to corruptions in data values.[1] If the page misses in the TLB that is software managed, the OS may be invoked to handle the fault, further activating the fault.

Corruption in Data Values

A corruption in a data value is the most fundamental way in which a hardware fault can affect the application. The data value corruption can affect the program execution in the following ways:

Control Instructions: If the fault propagates to a control instruction, either the conditional that decides the direction of the branch or the target of the branch may be corrupted.

If the conditional is affected without affecting the outcome (e.g., in $a < 0$, if the value of a changes from -5 to -10), the fault is logically masked by the application. If the outcome is altered, control branches to an alternate target, producing more faulty data values on the new path of execution.

If the target is affected by the fault, control may branch to either an invalid target, or to a valid legal target that is different from the correct (fault-free) target. An invalid target outside the code space results in an *Illegal Instruction* trap. Invalid targets to within the code space can either jump to the start or to the middle of a basic block. A subset of such branches, and branches to other legal targets can be identified using a *Hang Detector*. In either case, more faulty data values are produced by the erroneous branch, further propagating the fault.

Address of memory instructions: The outcome of a fault propagating to an address is similar to that of a fault that directly affects the address of a memory instruction (Section 6.1.1).

[1] Heap allocation by malloc() may not be contiguous, resulting in *holes* of unallocated addresses in valid pages.

Pure Data values: All other corruptions fall into this last category where only data values that are neither used in control flow nor address computations are affected. These faults are the hardest to detect as these are data-only corruptions. The *Division-by-Zero* detector of SWAT identifies some of these faults. Extensions to SWAT that include compiler-assisted invariant violation detectors [66], and other data-centric detectors [23, 44, 61] may be used to detect such faults.

Implications

From the above qualitative description of how faults propagate through the application, we see that SWAT has detectors for most outcomes except for faults that result in data-only corruptions. Despite this shortcoming, SWAT has large success in detecting hardware faults injected in several parts of the processor indicating that such behavior may be rare in real software.

6.2 Data-Only Values

We define a value to be a *data-only value* if it never affects a value that is used in a control instruction or a memory address. As previously described, the model of fault propagation through the application predicts that faults in such data-only values are harder to detect by SWAT and hypothesizes that since SWAT is highly effective to detect faults, such values should be far and few in the application. We take a closer look at these values here and study how faults in such values behave, as opposed to faults in other random application values.

6.2.1 Number of Data-only Values in Applications

Figure 6.2 presents the fraction of data-only values in the C/C++ workloads of the SPEC CPU 2000 benchmark suite. The data is collected from a simulated system running unmodified OpenSolaris OS on an UltraSPARC-III-like processor with a full memory hierarchy. The system is simulated with the Simics full-system simulator along with the GEMS timing models for processor and memory (Tables 3.1 gives the parameters of the processor and memory in the simulated system). For each workload, we pick 4 random phases (each 11 million instructions long) during the application execution and track with the help of a Dynamic Data Flow Graph (DDFG) how data values produced

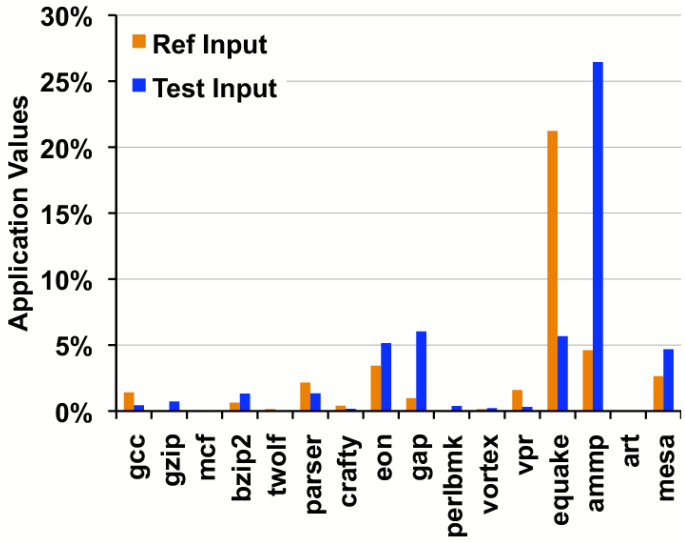


Figure 6.2: Fraction of application values that are data-only in the SPEC workloads. The small number of data-only values demonstrates that faults in most values propagate to control instructions and/or memory values, resulting in detectable anomalous execution. Further, workloads with floating point data have more data-only values making them more vulnerable than those that purely operate on integer values.

in the first million instructions in each phase are used in the next 10 million instructions of the program (dependencies through memory stores and loads are also tracked in the DDFG). If a value does not affect control instructions or memory addresses in this 10 million instruction window, it is classified as a *data-only value*. We do not track the values all the way until application output as the DDFG grows to unmanageable sizes for longer simulation periods (of billions of instructions); our results are therefore conservative as the set of values we classify as data-only is a super set of the true data-only values. The figure aggregates the number of data-only values across the four phases for each workload. The fraction of values that are data-only are shown for the `ref` and `test` inputs of SPEC in order to gauge the effects of changing the size of the inputs on the number of the data-only values.

From this figure we see that the fraction of values that are data-only is fairly low across the entire suite of SPEC workloads for both the larger `ref` and the smaller `test` input sets. Nearly all the workloads have $< 6\%$ of values as data-only for both sets of inputs. Workloads that operate on floating point values (such as *eon*, although it is a SpecInt benchmark, *equake*, *ammp*, and *mesa*) have more data-only values than the other workloads as floating point values are seldom used for

control decisions.

The three outliers from this observation are the FP workloads *art* for both sets of inputs, *equake* for the `ref` input, and *ammp* for the `test` input. *art* has no data-only values in the measured intervals as it takes several control decisions based on floating point values, unlike other FP workloads. (This workload tries to recognize known objects in a given image, resulting in elaborate FP-value dependent control-flow.) On the flip-side, the fraction of data-only application values is $> 20\%$ for *equake* with the `ref` input set (with 21% data-only values), and *ammp* with the `test` input set (with 26% data-only values). In our chosen phases, these configurations exhibit half an order of magnitude or more floating point operations than the other benchmarks, resulting in much higher data-only values. We believe that these outliers are artifacts of our simulation not tracking data-only values for the entire application, and are not fundamental to these workloads.

Additionally, we see no clear relationship between the size of the inputs and the fraction of data-only values. Although the larger `ref` input has more data values because of larger arrays and data-structures, it does not have more data-only values than the smaller `test` input. One possible reason for this is that the data-only value categorization also depends on the extent of control flow in the application which varies based on the inputs provided. The `ref` input, for example, may have more extensive control flow, making many of its data values affect control operations or memory addresses, resulting in fewer data-only values overall. Data-only values is thus a property of the application and is not necessarily related to the sizes of the inputs provided to the application.

The overall low fraction of data-only values demonstrates why software anomaly detection schemes like SWAT are effective. Faults in a large fraction of the values eventually affect control decisions or memory addresses, resulting in anomalous software execution such as branching off to a wrong location or indefinitely looping because of a fault in the control instruction, or out-of-bounds accesses and protection violations from faults in memory addresses. Since software anomaly detectors, like SWAT, monitor the software for such behaviors, they detect a large fraction of the hardware faults.

In order to understand the effects of infrastructure limitations to track data-only values, Figure 6.3 presents how the fraction of application values that are classified as data-only changes as we

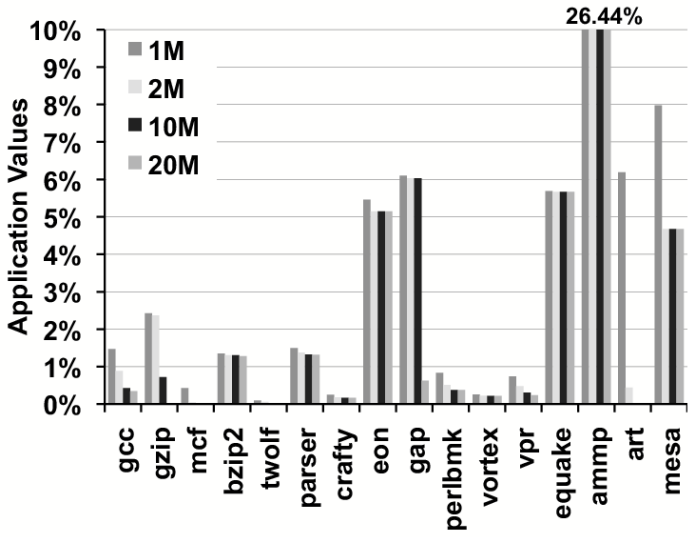


Figure 6.3: Number of data-only values with increasing window of propagation for SPEC workloads with `test` inputs.

track the uses of the values for intervals of 1M, 2M, 10M, and 20M instructions. (Figure 6.2 only showed the results for a 10M instruction propagation window.) The results are shown only for the `test` input as we observed similar results from the `ref` inputs. The only application with $> 10\%$ of application values as data-only (the scale of the y-axis is from 0%–10%) is *ammp* which has 26.4% data-only values, consistent with Figure 6.2.

The figure shows that, as expected, the fraction of values classified as data-only decreases as we increase the window of propagation. This is because with longer windows, the value affects other values which may eventually be used in control decisions or as memory addresses. We also see from the figure that tracking the propagation for a longer window of 2M instructions, instead of a shorter 1M instruction window results in the largest reduction of data-only values (an average reduction of 17%). Increasing this window, however, reduces the overall data-only values count by smaller fractions, with rather small differences between propagation windows of 10M and 20M instructions (only 3 applications show any differences of $> 2\%$ in the fraction of data-only values between these windows).

Therefore, although we consider a non-ideal propagation window and do not track the uses of the values till the application output, these results show that the limitations of our infrastructure

may not result in significant errors in the categorization.

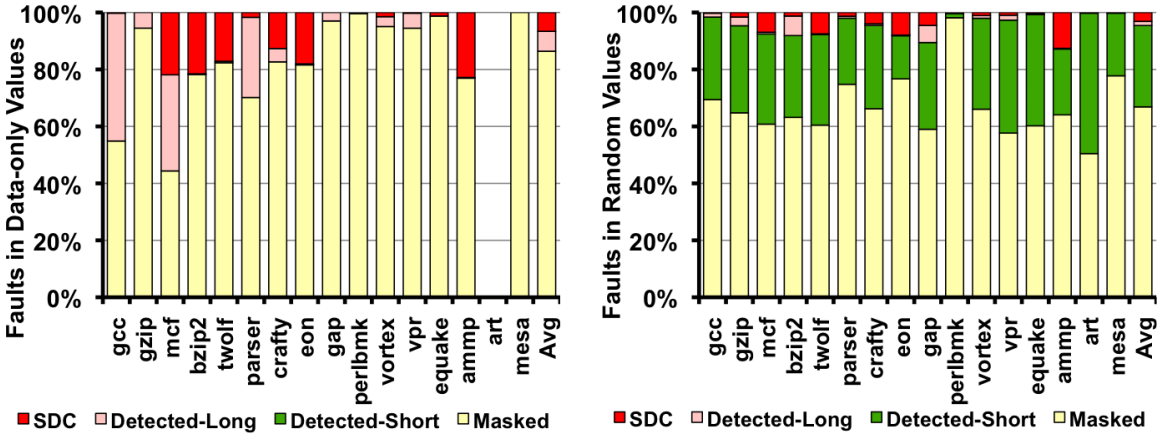
6.2.2 Faults in Data-only Values

Since these data-only values may not affect control decisions or memory addresses, faults in such data-only values may not result in visible anomalous software execution, making them harder to detect than faults in randomly chosen values.

Figure 6.4 compares the outcomes of injecting transient faults into (a) data-only values, and (b) random application values for all 16 SPEC C/C++ workloads with the `test` inputs.[2] Each workload is simulated in the afore-mentioned simulation framework and a transient fault is injected (one per run) into a random bit in the value chosen in the first million instructions of each phase. In Figure 6.4(a), up to 4000 randomly chosen values from the set of data-only values identified for a propagation window of 10M instructions (Figure 6.2) are chosen for injections. The workloads *mcf* and *twolf* have only 255 and 561 data-only values in the measured window, respectively, resulting in one injection in each of their data-only values. For Figure 6.4(b), 4000 random values that form the destination values of the first million instructions are chosen for injections. Once a fault is injected, the system is simulated for 100K instructions in detailed timing mode to identify faults that are detected by the SWAT detectors (Chapter 3) at short latencies. Faults detected in this window are classified as *Detected-Short*. (We chose 100K as the threshold for this category based on the fault recovery results presented in Chapter 5.) Faults that are are not detected in the 100K instruction window are then simulated functionally until either the application finishes or the fault is detected. Faults detected in this mode are classified as *Detected-Long* category and are classified as unrecoverable. Activated transient faults that produce application outputs that are identical to the fault-free output are classified as *Masked*, while those that corrupt the outputs are classified as *SDC*. The large number of fault injections give a standard error of under 0.2% for the SDC rate at a confidence interval of 95%.

Faults that are either detected at longer latencies or result in SDCs are hard to detect by the SWAT detectors, making it important to protect those values through special techniques. From

[2] We chose the smaller `test` inputs over the larger `ref` inputs in order to contain the duration of the fault injection experiments.



(a) Data-Only Values **(b) Random Values**

Figure 6.4: Outcome of architecture-level transient faults injected into (a) data-only values and (b) random values for SPEC workloads. Faults in data-only values are detected at longer latencies, and lead to more SDCs, making such values critical to protect to harden the application against hardware faults.

Figure 6.4(a), we see that on an average, 6.8% and 6.5% of the faults in data-only values fall under the *Detected-Long* and *SDC* categories, respectively. The corresponding numbers from the faults in random values shown in Figure 6.4(b) are, on the average, only 1.4%, and 3.1%. Faults in data-only values may cause detectable software anomalies when the values affect control instructions and/or memory operations. Since our infrastructure does not track the propagation until application output, as previously discussed, it categorizes some of these faults as data-only. Nevertheless, since all the faults detected in data-only values are detected at long latencies, they are arguably as critical to protect as those that result in SDCs.

We also see that several applications show marked differences between faults in data-only value and random values, with some apps demonstrating high SDC rates from data-only values (the *ammp* workload, for example, has an SDC rate of 22.8% in Figure 6.4(a)). This demonstrates the importance of considering application-specific properties when studying the effect of faults.

Figure 6.4 shows a couple of other interesting trends in fault masking and fault detection. First, we see that the masking rate is higher for faults in data-only values than for faults in random values; on the average, 85% of the faults in Figure 6.4(a) are masked while only 66% of the faults in Figure 6.4(b) are masked. Faults in random values show lower masking rates as the values

affected by the fault may be used for a variety of operations including control-flow and memory addressing. Faults in data-only values, on the other hand, are used purely for data-flow and have a higher chance to be logically masked.[3] Second, we see that over 86% of the unmasked faults in random values are detected in under 100K instructions by the SWAT detectors (*Detected-Short in Figure 6.4(b)*), demonstrating that the SWAT detectors are highly effective in detecting faults in random application values. In Figure 6.4(a), however, only 2.5% of the unmasked transient faults in data-only values are detected at short latencies showing that data-only values need additional support to enable software anomaly detection.

These results show that faults in data-only values are hard-to-detect with existing software anomaly detectors. Additional support from the application may be required to enable software anomaly detection for faults in such values.

6.2.3 Detecting Hard-to-detect Faults in Data-Only Values

The existing SWAT detectors are therefore insufficient to detect faults in data-only values. Over 13% of the faults injected into data-only values in Figure 6.4(a) were classified as *Detected-Long* or as *SDC* and are thus hard-to-detect. We subsequently leverage existing work on using value-based software-level invariants to detect such hard-to-detect faults [66].

For each application, we derive invariants on the dynamic values of all static instructions that have at least one data-only dynamic instance.[4] The invariants track the minimum and maximum values produced by dynamic instances of the chosen instructions and enforce the ranges at run-time. Table 6.1 gives the number of data-only locations in each workload at which such value-based invariants are derived and monitored.

For each instruction, the range of values monitored by the invariants are derived across the 4 randomly chosen phases in the SPEC workload and are aggregated per static instruction across the four phases. The invariants are then monitored within the afore-mentioned simulation framework for fault injection without modifying the application. If a static instruction that contains an invariant

[3] Note that the fault model used here is an instruction-level fault model, as opposed to the microarchitecture-level fault model used in previous chapters, resulting in far more activations than those seen previously.

[4] We saw in our experiments that not all dynamic instances of a given static instruction had identical behavior, resulting in some being classified as data-only while others were not.

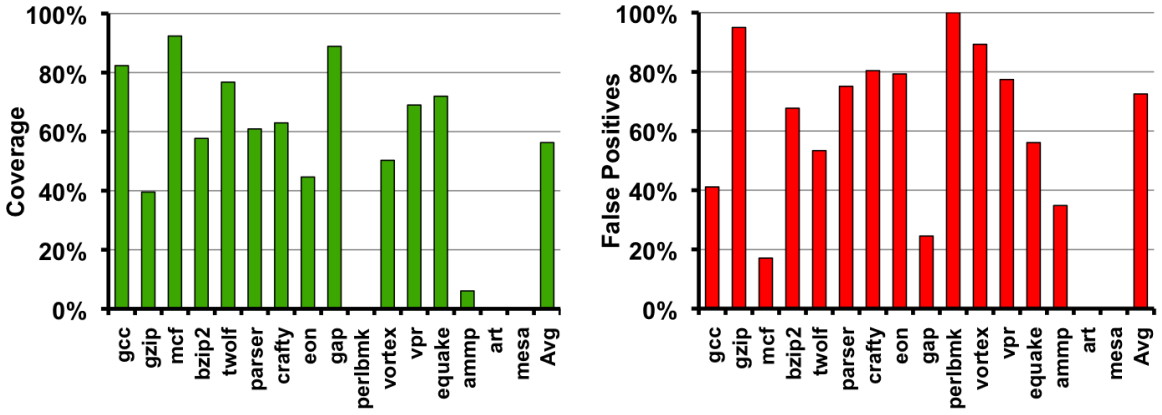
Application	Number of Locations
gcc	41
gzip	2
mcf	12
bzip2	10
twolf	15
parser	49
crafty	55
eon	65
gap	13
perlbmk	5
vortex	38
vpr	17
equake	17
ammp	100
art	0
mesa	0

Table 6.1: Number of static locations chosen for inserting value-based invariants to detect hard-to-detect faults in data-only values. Each chosen location has at least one dynamic instance that was classified as data-only.

attempts to retire, the dynamic value produced is checked against the ranges monitored by the invariant – if the value does not conform to the range, then a detection is flagged. In order to explore the limits of these invariants, we use the same set of inputs (the `test` inputs for SPEC) to derive the invariant ranges and to enforce them.

Figure 6.5 shows the effect of using of these invariant detectors to detect the hard-to-detect faults in data-only application values. Figure 6.5(a) shows the fraction of hard-to-detect faults that are detected by these range-based detectors. Since the application may tolerate some value perturbations that makes the dynamic values violate the monitored ranges, these invariants may also incur false positives. Figure 6.5(b) shows the fraction of faults detected by these detectors that would have been masked by the application were the execution allowed to continue. Each graph also shows the results aggregated across all the workloads in the *Avg* bar.

We see from Figure 6.5(a) that, on the average, 56% of the hard-to-detect faults are detected by these detectors. This translates to reducing the rate of hard-to-detect faults from 13.8% to 6.5%. Although not shown in this figure, these detectors cover 69% of the faults in data-only values that are detected at long latencies (*Detected-Long* in Figure 6.4(a)) and 39% of the SDCs in data-only



(a) Coverage of Hard-to-Detect Faults **(b) False Positives Rate**

Figure 6.5: Effect of using range-based invariant detectors to detect hard-to-detect faults in data-only application values.

values. In most workloads, a significant fraction of the hard-to-detect faults are identified, except for *perlbmk* where no hard-to-detect faults are detected and *ammp* where only 6% of the hard-to-detect faults are detected. Further sophistication in the types of detectors used to identify the remaining faults is required.

Figure 6.5(b) shows that these detectors also incur a high rate of false positives. On an average, 72% of the faults detected would have been masked by the application were the execution allowed to continue, making them false positives. Recent work has however shown that training such value-based invariants on multiple inputs and altering the ranges dynamically based on the observed false positives reduces this false positives rate significantly, with the rate becoming nearly 0% when training with 50 inputs [66, 80]. Such a strategy may be used to reduce the false positives incurred from these detectors. Further, since such faults in data-only values are rare (given that $< 6\%$ of application values are data-only), these false positives may not cause notable perturbations when these detectors are deployed in commodity systems.

These results show that range-based value invariants may be effective to detect hard-to-detect faults in data-only values. However, further research is required to identify the types of faults that escape these detectors and to reduce the false positive rates incurred. Such explorations are beyond the scope of this thesis and is left to future work.

6.2.4 Detecting Hard-to-detect Faults in Random Values

Although detecting faults in data-only values is important to improve the efficacy of software anomaly detection, faults in other random values that escape the SWAT detectors are also important to detect. In fact, from the faults injected in random values in Figure 6.4(b), only 12% of the hard-to-detect faults are from data-only values. The remaining 78% are from faults in other values that do affect control instructions and memory addresses and are hence not data-only.

We identified through some analysis that these faults tend to have some close connections with the data-only values that have been explored previously. Although the fault corrupts values that affect control instructions and/or memory addresses, we noticed that the values affected as a result of the control flow deviations tended to be data-only values. The control flow subsequently merges with that of the fault-free execution, resulting in no software-level anomalies (previous work has demonstrated that such control convergences in the presence of faults that affect branches are common occurrences in real-world applications [77]). Effective detectors that detect faults in data-only values may thus also detect hard-to-detect faults in random values. We however leave the exploration of such detectors to future work but believe that the insight presented here would help guide the development of such detectors.

6.2.5 Identifying Critical Data-Only Values

Detecting faults in data-only values may not only reduce the hard-to-detect faults from data-only values but may also help reduce the fraction of hard-to-detect faults in random values. We therefore analyze the data-flow properties of the data-only values in an attempt to predict which values are more important to protect than which others. While there has been much work to identify such critical application values, we explore the previously proposed metric of fanout [56] owing to it overwhelming success in identifying critical application values.

The fanout of a value is defined as the number of dynamic instructions dependent on, i.e. uses the value as a source, the static instruction that produces this value. Since a value with a high fanout will result in propagating the fault to more values, it is classified as being more critical to protect than one with a low fanout. We use this notion of fanout to identify which critical data-only

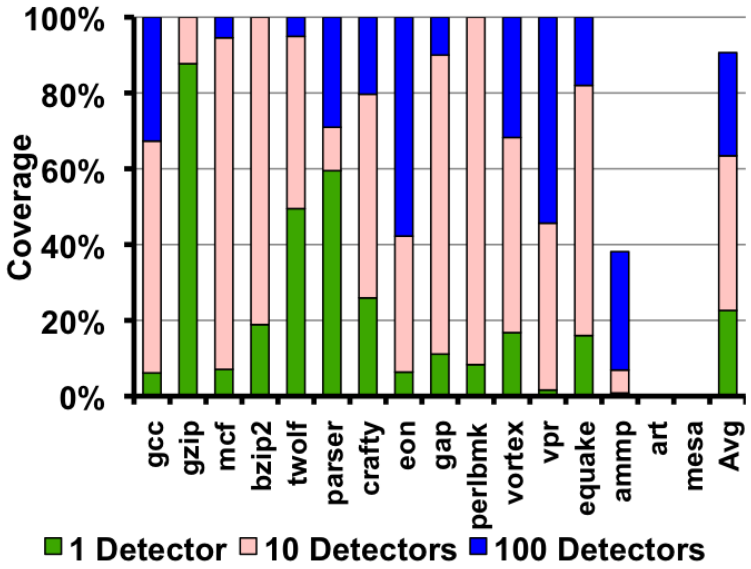


Figure 6.6: Efficacy of oracular detectors derived with the fanout metric to identify hard to detect faults. The fanout metric identifies critical values to protect accurately. However, the detectors must be carefully designed to keep the false positive rates low.

values need to be protected so as to reduce instances of hard to detect faults. For each data-only value, the fanout is computed by aggregating (with set union) all dynamic instructions that use as source a data-only dynamic instance of the corresponding static instruction.[5]

We select those data-only values that have at least one hard-to-detect fault and rank them based on their fanout. We then select the top 1, 10, and 100 data-only locations and assume oracular predictors that identify the presence of *any* fault at these data-only values. Figure 6.6 shows the effect of deploying these oracular detectors in the SPEC workloads. For each application, the stacks show the fraction of hard-to-detect faults in data-only values that are identified with 1, 10, and 100 oracular detectors placed based on the fanout metrics. The results aggregated across all the workloads is also shown in the *Avg* bar.

From the figure we see that a small number of detectors, selected based on the fanout metric may be sufficient to detect a large fraction of the hard-to-detect faults. Since values with high fanout affect much of the application state, faults in these values affect the application in noticeable ways. With just 10 detectors, Figure 6.6 shows that over 80% of the hard-to-detect faults in several

[5]Previous work aggregated fanout statistics across *all* instances of a static instruction with the assumption that multiple dynamic instances may share identical behavior in the presence of faults. We have found, however, that this need not be the case and hence aggregate our statistics only for those instances that produce data-only values.

workloads are detected, with an average coverage of 62% of the hard-to-detect faults. With 100 detectors, all the hard-to-detect faults in data-only values are covered for all workloads except *ammp*. *ammp* exhibits a coverage of only 38% with even 100 detectors as it has over 500 static values that cause hard-to-detect faults (recall from Figure 6.2 that this application had the maximum number of data-only values for the `test` input set); increasing the number of detectors for this workload would improve its coverage with 100% coverage at approximately 500 detectors. The workload *art* has no data-only values and all faults injected into data-only values in the workload *mesa* are masked, resulting in no detectors for these two workloads.

The fanout metric may thus be used to select effective locations for software-level detectors to identify the hard-to-detect faults. Although we leave the exploration of the specific detectors to future work, we would like to note that an important consideration for these detectors is to keep the false positives from the detectors under check. Of the faults in the top-10 locations chosen for detectors for each application based on the fanout metrics also, 69% were masked by the application on average. The detectors would thus have to be carefully chosen to not identify such faults and to keep the false positives under check.

6.3 A Hardware-Centric View of Faults

Another axes to characterize the properties of these hard to detect faults that may result in SDCs is a hardware centric view. Understanding SDCs from this view will help identify those structures in which faults are harder to detect, guiding the choice of which structure should be hardened against hardware faults.

6.3.1 Structure-specific SDC Rates

Section 3.3.1 presented the potential SDC rates and detection latencies for faults injected in several microarchitecture units in a modern processor. From the discussion in that section, we saw that the ability of software anomaly detectors to detect hardware faults depended on how the hardware fault manifested at the application-level instruction. Faults in structures that affect control flow in the program (such as the Decoder, ROB, and RAT) are the easiest to detect as they result in

anomalous software execution identifiable at low latencies. Faults in structures that affect data-flow (Int ALU, Reg Dbus, Int Reg, AGEN, and FPU) tend to have higher rates of potential SDCs than the former 3 structures. However, barring permanent faults in the FPU, most faults in even these structures were detected. This is because a large fraction of the values corrupted from faults in these structures also affect values that are used in control decisions and/or memory addresses, resulting in anomalous executions that the SWAT detectors monitor. Faults in structures such as the FPU that purely affect data flow without affecting control (analogous to the data-only application values that were previously discussed), are the hardest to identify, making these structures prime candidates for hardware techniques to hardware against faults. There have been several techniques proposed to harden such structures against faults, such as residue coding [37], which may be applicable to further reduce the SDC rate.

Building a model that accurately captures this manifestation and predicts which application-level instruction would be affected by a hardware fault would be immensely useful to develop cross-layer resiliency solutions that achieve the highest reliability targets at the lowest cost. However, building such models may not be straightforward as faults in some hardware units may manifest themselves in not-so-direct ways at the application level, making it hard to establish the link between a hardware fault and its application-level manifestations. Faults in the cache tag array, privileged bits are some such examples. We leave exploring such models to future work beyond the scope of this thesis.

6.3.2 Effect of Utilization on SDC Rates

Since the hardware of modern processors have replicas of many units to increase performance (such as multiple decoders, multiple ALUs, and multiple address-generation units), faults in less-utilized structures may see lower activation rates, resulting in subtle corruptions of the software and in higher SDC rates. Although prior work has identified that this lowered activation rate may make fault diagnosis harder [71], the effect of altering this activation rate on the SDC rate has not been studied.

Table 6.2 shows the effect of enforcing a uniform round-robin utilization on the SDC rates for permanent and transient faults in 3 hardware structures – the decoder, the Integer ALU, and the

Fault Type	Non-Uniform	Uniform
Permanents	57	54
Transients	27	24

Table 6.2: Reduction in Potential SDCs in 3 structures from uniform round-robin utilization for permanent and transient faults.

address generation unit. The results are obtained by injecting 1280 permanent and 1280 transient faults in each structure while running all 16 C/C++ SPEC CPU 2000 workloads within the afore-described simulation environment. For each workload, we pick four random phases during the application's execution, inject faults in them, and perform detailed timing simulation (with the GEMS timing models) for 10M instructions to detect the fault with the SWAT detectors. Faults that are not detected in this window are simulated functionally (with just simics) to completion and those faults that affect the output of the application and are not detected are classified as Potential-SDCs, as the application may tolerate some of these faults in its output. The table shows the *Potential SDCs* from the base system that uses non-uniform scheduling for these 3 structures, and the one that uses uniform round-robin scheduling. In the simulated base system, there are 4 instances of each of these units, with the first instance being preferred over the second (which is used only when the first instance is currently busy) and so on. In the system that uses uniform round-robin scheduling, the instructions are scheduled to the 4 units in a round-robin fashion by remembering which instance was last used.

From Table 6.2, we see that uniform scheduling reduces the instances of SDCs by 5% for permanent faults and by 11% for transient faults. Round-robin scheduling is therefore useful to reduce the potential SDC rate by making the faults more visible to software.

6.4 Summary and Implications

This chapter builds the intuition behind why the SWAT detectors are effective in detecting hardware faults, and yield low rates of silent data corruptions. It builds this intuition along two angles, one with an application-centric view and another with a hardware-centric view. With an application-centric view, it demonstrated that faults in most application values affect control operations and/or

memory addresses, resulting in identifiable anomalous software execution. Only a small fraction of application values do not affect control operations or memory addresses, making the SWAT detectors effective for handling a majority of the faults that affect modern workloads. This chapter also shows that faults in these *data-only values* are hard-to-detect and need special software-level detectors for fault detection. (Since the software may be more adept at detecting faults in pure data values through built-in consistency checks, and the hardware is more adept at detecting perturbations in control flow or memory addresses, this is arguably the right distribution of specialized detectors based on the manifestation of the hardware fault at the application level.) These software-level detectors may be placed with the help of data-flow properties of the application. Second, taking a hardware-centric view, this chapter showed that faults in certain structures that influence the control flow of the application are easier to detect than those in structures that are data-centric. It also shows that a uniform hardware schedule of redundant microarchitecture units may help lower the SDC rates by ensuring that faults in rarely used structures are activated more often.

The results presented in this chapter may be used to design fault-tolerant applications and hardware that is less prone to failures. The data-only categorization of application values may be used to guide which application-level data values need to be protected in order to keep the SDC rates under check. These data values can be protected through several techniques that have been proposed to protect applications from faults in data values [24, 61, 66]. The categorization also exposes those values in which faults are hard-to-detect and can be used to guide fault injection techniques to stress-test future fault tolerant solutions. Additionally, the hardware-centric analysis helps select which structures to harden against hardware faults.

Chapter 7

Conclusion and Future Work

7.1 Summary and Conclusions

The problem of unreliable hardware is expected to affect commodity systems of the future, warranting alternatives to traditional redundancy based solutions that incurred high overheads. Such solutions should incur low-cost in performance, power, and area while detecting, diagnosing, and recovering the system from hardware failures. Although there have been several proposals to implement low-cost solutions for detection, recovery, and diagnosis, most of them are piece-meal solutions, making the cumulative cost of resiliency unacceptable.

This thesis presents SWAT, standing for SoftWare Anomaly Treatment, which takes a full-system approach to system resiliency and presents a solution to detect and recover from in-core permanent and transient faults. (SWAT also implements an effective strategy for fault diagnosis [28, 34] to identify the root-cause of the fault but is not discussed in this thesis.) SWAT detects hardware faults by observing anomalous software execution, ignoring faults masked at lower levels of the hardware and system. Consequently, SWAT relies on checkpointing-based schemes for rollback recovery, and buffers externally visible outputs and releases them only when they are known to be fault-free (thus handling the *output commit problem*). The SWAT diagnosis module leverages support from this checkpointing system to repeatedly replay the faulting execution to distinguish between transient faults, software bugs, and permanent hardware faults in single- and multi-core systems. This thesis also presents an intuition behind why this strategy is effective for fault tolerance by presenting an in-depth characterization of how faults propagate through an application, and by studying the types of values affected by the fault.

Our evaluations show that the SWAT strategy is effective to detect and recover from in-core

hardware faults. (Interested readers are referred to other work that demonstrates the effectiveness of the SWAT diagnosis module [28, 34]). In particular, the evaluations in the thesis demonstrate the following.

- The low-cost SWAT detectors incur near-zero performance, area, and power overheads and detect a high fraction of the permanent and transient faults injected into various microarchitecture structures in a modern processor with compute-intensive and I/O-intensive applications. Under 0.6% of the faults in non-FPU structures escape the SWAT detectors and corrupt the output of the application. By considering the operations of the application, we identify that a large fraction of these erroneous outputs may actually be tolerated by the applications, resulting in only 0.2% of the injected faults resulting in SDCs. This translates to a two orders of magnitude reduction in the FIT rate when compared to the base system with no protection against faults (all faults that are not masked lead to failures in the base system). Structures such as the FPU that affect only data computations need additional support for software anomaly detection.

- The evaluations from fault recovery show that buffering external outputs in software may still make the outputs vulnerable to faults in the processor core. This thesis presents, for the first time, an implementation and evaluation of buffering external outputs in hardware and demonstrates that such a strategy may be used if the checkpoint interval from the detectors is under millions of instructions (or sub-millisecond range in a modern processor). Since the SWAT detectors are adept at detecting hardware faults at such short latencies, we show that $> 94\%$ of the harwdare faults may be detected and recovered by SWAT at a short checkpoint interval of 100K instructions. At this checkpoint interval, fault-free execution is minimally affected from output buffering; the performance impact is $< 5\%$, and the area overheads are under 2KB.

- This thesis also presents an intuition behind why the SWAT strategy is effective for handling faults in modern workloads. It shows that only a small fraction of the values in modern workloads do not affect control decisions or memory addresses, making faults in majority of

the values cause deviation in control flow or memory addresses. Since SWAT is effective in detecting such perturbations in the system, it detects a majority of the hardware faults. The thesis also shows that faults in such data-only values are hard to detect, warranting additional support from the software to render them amenable to software anomaly detection.

The above findings have far reaching implications on resilient systems design. First, it makes the case for software anomaly detection to be used in future systems by demonstrating that they are highly effective, incur low cost during common mode of fault-free operation, and incur short detection latencies that enables low-cost fault recovery. Second, it shows that considering application-level properties is important when evaluating the fault tolerance of software anomaly detection schemes. Ignoring this aspect of fault tolerance may lead to unnecessarily conservative estimates about their fidelity. Third, it presents a strategy for hardware output buffering while incurring low performance and area overheads, and with no changes to excising device hardware. Finally, it performs a first-cut evaluation of the fundamental reasons behind why software anomaly detectors, such as the ones used in SWAT, are effective and identifies values which may require additional support for software anomaly detection.

7.2 Limitations and Future Work

The ideas and evaluations presented in this thesis are not without assumptions and limitations. In this section, we highlight some limitations of the work presented in this thesis and suggest directions for future research on those topics.

7.2.1 Leveraging Cross-Layer Support to Lower the SDC Rate

The SWAT system presented in this thesis achieves low SDC rates for compute-intensive and I/O-intensive workloads. However, this rate may not be low enough for certain classes of applications such as banking or mission critical applications. For such applications that require attaining higher reliability targets and for other systems that require even lower cost for the achieved reliability targets, support from the application or other venues may be required.

To this end, there has been a recent thrust towards using cross-layer techniques that span solutions from the hardware-level all the way up to the application-level to reduce instances of such SDCs at the lowest possible cost [18]. While SWAT is a major step in this direction, there are plenty of alternatives to consider for this purpose. An important paradigm to consider when exploring this direction of research is to understand the trade-offs between performance overheads and resiliency targets that such detectors are capable of achieving; exploiting these trade-offs would be crucial to achieve the highest resiliency targets at the lowest possible overheads.

Using application-level assertions and consistency checks embedded by the programmer into the application is one such strategy to achieve lower SDC rates at a marginally higher cost. Although many of these assertions are turned-off for production runs, modern software has an increasing number of these assertions left on even in production codes to eliminate instances of unexpected outcomes from executions that were not previously tested [81, 82]. Since these checks are inserted by the programmer, they may be highly effective as they encapsulate the semantics of the application. In addition to such checks, there is a plethora of literature on techniques to identify corruptions in application values, with specific focus on those corruptions that affect pure data values (the *data-only* classification presented in Chapter 6). Exploiting these assertions to improve the resiliency folds well into SWAT as one it its philosophies involves the use of software support to improve the resiliency against hardware faults. The iSWAT framework, which used compiler-assisted likely program invariants for hardware fault tolerance, is a small step in this direction.

7.2.2 Low-Cost Fault Recovery for Multithreaded Applications in Multicore Systems

Multithreaded applications running on multi-core systems is emerging as the configuration of future systems. The threat of unreliable hardware is expected to plague even such systems, warranting low-cost solutions to detect, diagnose, and recover from faults in such systems. The mSWAT extension of the SWAT framework presented in this thesis demonstrated that the SWAT strategy is applicable for detecting and diagnosing faults in such multicore systems when running multicore workloads [28].

While some aspects of fault recovery in multi-core system have been explored, there are two aspects that warrant special attention. The first aspect is enabling fault recovery with multi-core systems in the presence of I/O. Extending the notion of hardware output buffering presented in this thesis to multicore systems it not straight-forward because the buffers in the different cores would have to be synchronized with each other in-order to keep the global order of I/O operations consistent (mechanisms similar to those proposed by SafetyNet [71] or ReVive [60] for memory consistency in multicore systems may be leveraged). Further research is required to implement effective solutions that incur low performance, and area overheads. The second aspect of implementing recovery for future systems is to reduce the overall area overheads incurred by the operations involved in checkpointing. Although SafetyNet has demonstrated a low-cost strategy to reduce the performance overheads from checkpointing in multicore systems, the area overheads from storing the memory logs is of the order of 100s of KBs [71]. We propose a technique where only a part of this overhead is incurred in hardware (Section 5.3.1). The overheads may, however, be further minimized by taking additional support from the software to infer the minimal state that is required for restarting the executions. There have been several recent proposals on low-cost fault recovery that lower the amount of state to checkpoint by understanding the functionality of the application which may be leveraged for this purpose as well [1, 2, 10, 25, 32, 36, 69]

7.2.3 Analytical Model of Application Resiliency

Chapter 6 presented a first-cut understanding of how faults propagate through an application and builds an intuition behind why the SWAT detectors are effective in detecting hardware faults with such high fidelity. This is helpful in understanding the relationship between applications and faults, and identifies an important class of application values (*data-only values*) that need additional support for software anomaly detection. It would however be helpful to convert this into an analytical model which, given an application a fault model, and a set of detectors, *predicts* the SDC rate of the application with high fidelity. Such a model would help do away completely with fault injection experiments, and would help devise both fault-tolerant applications and detection schemes which are highly effective. We believe that the concepts presented in this thesis, along with the tree of

application-level fault propagation (Figure 6.1) lay the foundations for future research to develop such analytical models. Research for such an analytical model is already under-way in the SWAT group [29].

7.2.4 A SWAT Prototype

The evaluation of SWAT presented in this thesis is largely within a simulation framework that simulates a full system running a modern operating system with peripheral I/O devices. Although this evaluation has high fidelity (based on the comparison of SWAT's results to results from faults injected with lower gate-level simulators [33]), a prototype of SWAT demonstrating its abilities to detect, diagnose, and recover faults is necessary to enable transfer of the SWAT technology to industry.

Implementing such a prototype would require solving real-world engineering problems in both software and hardware levels. At the software-level it involves implementing the actual SWAT firmware that orchestrates the diagnosis and recovery process once a fault is detected. It also requires handling the complex interactions of the hardware with the operating system, which may require innovations at various levels. At the hardware-level, the prototype would demonstrate the true hardware overheads incurred by the SWAT detectors; although the detectors monitor software anomalies, they are implemented in hardware. The prototype would also require implementing the hardware module that achieves fault recovery and would demonstrate the applicability of SWAT's strategy for hardware output buffering to generic devices.

This prototyping of SWAT is already underway. Concurrent with this thesis, my colleagues have implemented a first-cut prototype of the SWAT system, and have demonstrated the ability of SWAT to detect gate-level stuck-at and transient faults injected in various gate-level modules [58]. There is however a long way to go to implement the diagnosis and recovery modules, and demonstrate that the robustness of the SWAT strategy to generic types of faults.

7.2.5 Other Fault Models and Faults in Other Components

SWAT has so far been evaluated for single stuck-at and transient faults for various components within the processor core. We have demonstrated in the SWAT-Sim work that these fault models may be inaccurate representations of faults at the gate-level, although the inaccuracies do not lead to differences of orders of magnitude [33]. Nevertheless, there is much work that can be done in the future to explore SWAT under other fault models and for faults in other components.

With respect to other fault models, there are several fault models emerging as the most representative model of faults in future systems. Process variations, NBTI degradation, intermittent faults, multiple faults are but some of the many such fault models that are emerging. While the representativeness of each of these models still constitutes avenues for future research, evaluating SWAT under each such fault model would help improve SWAT and make it a more robust solution for faults from several avenues.

Another aspect that needs to be evaluated is the fidelity of SWAT to detect faults in other components. The evaluations of SWAT presented in this thesis have been largely confined to faults within the core of the processor, and has not considered faults in other system components. Of particular importance are faults in off-core components such as caches and I/O-controllers as such faults are expected to become increasingly common with continued integration in the late CMOS era.

Appendix A

Data Tables For Graphs

Fault Type	Workload	Masked	Detected	Potential SDC
Permanents	SPEC	1965	6937	56
	Server	606	8295	16
Transients	SPEC	7520	909	58
	Server	7675	1239	37

Table A.1: Data for Figure 3.2 that shows the potential SDC rates from permanent and transient faults injected into non-FP units in server and SPEC workloads.

Permanent Faults

Type	Masked	Detect-App	Detect-OS	Potential SDC	SDC rate
Decoder	0	512	758	0	0.0%
INT ALU	21	398	858	2	0.2%
Reg Dbus	24	464	787	3	0.2%
Int reg	199	326	754	0	0.0%
ROB	0	647	629	0	0.0%
RAT	155	359	765	0	0.0%
AGEN	211	464	574	7	0.6%
Average	606	1526	5125	16	0.2%

Transient Faults

Type	Masked	Detect-App	Detect-OS	Potential SDC	SDC rate
Decoder	1136	83	54	7	0.5%
INT ALU	1212	49	11	8	0.6%
Reg Dbus	1188	51	35	6	0.5%
Int reg	1178	51	44	7	0.5%
ROB	1237	22	14	0	0.0%
RAT	540	227	510	1	0.1%
AGEN	1190	47	41	2	0.2%
Average	7675	207	709	37	0.4%

Table A.2: Data for server workloads in Figure 3.3 that shows the per-structure breakdown of the outcome of permanent and transient faults injected into server workloads.

Permanent Faults

Type	Masked	Detect-App	Detect-OS	Potential SDC	SDC rate
Decoder	295	768	214	1	0.1%
INT ALU	56	846	363	15	1.2%
Reg Dbus	58	909	295	18	1.4%
Int reg	307	742	220	11	0.9%
ROB	0	1240	40	0	0.0%
RAT	435	541	303	0	0.0%
AGEN	164	866	239	11	0.9%
FPU	953	5	100	222	17.3%
Avg No FPU	1315	5912	1674	56	0.6%

Transient Faults

Type	Masked	Detect-App	Detect-OS	Potential SDC	SDC rate
Decoder	1196	52	24	7	0.5%
INT ALU	1170	51	33	26	2.0%
Reg Dbus	1204	51	16	9	0.7%
Int reg	1230	23	17	10	0.8%
ROB	1237	34	1	1	0.1%
RAT	696	386	195	3	0.2%
AGEN	1224	31	23	2	0.2%
FPU	1279	0	0	0	0.0%
Avg No FPU	7957	628	309	58	0.6%

Table A.3: Data for SPEC workloads in Figure 3.3 that shows the per-structure breakdown of the outcome of permanent and transient faults injected into SPEC workloads.

Fault	Workload	$< 10K$	$< 100K$	$< 1M$	$< 10M$	$> 10M$
Permanents	SPEC	5821	6475	6710	6997	7053
	Server	6561	7170	7313	8195	8307
Transients	SPEC	543	701	765	828	884
	Server	563	715	834	1149	1254

Table A.4: Data for Figure 3.4 that shows the detection latency for permanent and transient faults in server and SPEC workloads.

Permanent faults

Fault	$< 10K$	$< 100K$	$< 1M$	$< 10M$	$> 10M$
Decoder	1073	58	16	117	16
INT ALU	1015	63	12	143	15
Reg Dbus	890	135	39	173	17
Int reg	609	204	54	181	14
ROB	1264	8	4	0	18
RAT	956	55	10	100	15
AGEN	754	86	8	168	17
FPU	0	0	0	0	0
Avg	6561	609	143	882	112

Transient faults

Fault	$< 10K$	$< 100K$	$< 1M$	$< 10M$	$> 10M$
Decoder	95	6	5	20	16
INT ALU	21	12	3	16	14
Reg Dbus	44	4	7	19	17
Int reg	22	10	3	35	13
ROB	35	0	0	1	14
RAT	284	118	89	219	16
AGEN	62	2	12	5	15
FPU	0	0	0	0	0
Avg	563	152	119	315	105

Table A.5: Data for server workloads in Figure 3.5 that shows the per-structure breakdown of the detection latency for permanent and transient faults in server workloads.

Permanent faults

Fault	$< 10K$	$< 100K$	$< 1M$	$< 10M$	$> 10M$
Decoder	894	47	27	19	8
INT ALU	946	124	49	65	4
Reg Dbus	949	119	60	57	10
Int reg	701	153	29	59	6
ROB	630	9	4	0	8
RAT	759	54	15	17	8
AGEN	892	123	35	36	4
FPU	50	25	16	34	10
Avg	5821	654	235	287	56

Transient faults

Fault	$< 10K$	$< 100K$	$< 1M$	$< 10M$	$> 10M$
Decoder	55	1	3	3	10
INT ALU	52	11	3	5	11
Reg Dbus	48	8	3	3	6
Int reg	41	22	2	7	7
ROB	22	0	0	0	6
RAT	289	106	53	44	8
AGEN	36	10	0	1	7
FPU	0	0	0	0	1
Avg	543	158	64	63	56

Table A.6: Data for SPEC workloads in Figure 3.5 that shows the per-structure breakdown of the detection latency for permanent and transient faults in SPEC workloads.

Fault	App-only	System	None
Decoder	524	865	1
INT ALU	381	904	0
FPU	0	0	0
Reg Dbus	384	927	0
Int reg	284	833	1
ROB	468	425	419
RAT	497	1188	146
AGEN	374	723	0
Average	2912	5865	567

(a) Server Workloads

Fault	App-only	System	None
Decoder	848	201	0
INT ALU	832	423	0
FPU	68	57	0
Reg Dbus	914	333	0
Int reg	716	296	2
ROB	428	16	221
RAT	713	511	113
AGEN	899	230	4
Average	5418	2067	340

(b) SPEC Workloads

Table A.7: Data for Figure 3.7 that shows the software components corrupted for faults detected in (a) server and (b) SPEC workloads.

Latency	Hard-Latency	Soft-Latency
$< 10K$	0.7910	0.8996
$< 100K$	0.8644	0.9336
$< 1M$	0.8816	0.9704
$< 10M$	0.9879	0.9932
$> 10M$	1.0000	1

(a) Permanent Faults

Latency	Hard-Latency	Soft-Latency
$< 10K$	0.4544	0.8711
$< 100K$	0.5771	0.8922
$< 1M$	0.6731	0.9244
$< 10M$	0.9274	0.9468
$> 10M$	1.0000	1

(b) Transient Faults

Table A.8: Data for the server workloads in Figure 4.7 that shows the distinction between Hard- and Soft- Latency for detected permanent and transient faults.

Latency	Hard-Latency	Soft-Latency
$< 10K$	0.7090	0.9216
$< 100K$	0.9246	0.9611
$< 1M$	0.9562	0.9752
$< 10M$	0.9928	0.9934
$> 10M$	1.0000	1

(a) Permanent Faults

Latency	Hard-Latency	Soft-Latency
$< 10K$	0.6122	0.8538
$< 100K$	0.7903	0.9061
$< 1M$	0.8625	0.9274
$< 10M$	0.9335	0.937
$> 10M$	1.0000	1

(b) Transient Faults

Table A.9: Data for the SPEC workloads in Figure 4.7 that shows the distinction between Hard- and Soft- Latency for detected permanent and transient faults.

Chkpt	apache	sshd	squid	mysql
10K	1.00	1.01	1.00	1.01
100K	1.00	1.01	1.04	1.01
1M	1.57	1.20	5.46	1.50
2M	3.19	1.41	11.69	2.13
5M	15.24	2.38	31.33	3.92
10M	18.99	4.09	62.62	7.39

(a) Performance overheads

Chkpt	apache	sshd	squid	mysql
10K	0.208	0.32	0.22	0.20
100K	0.972	1.51	1.99	1.49
1M	15.55	2.58	7.71	1.58
2M	15.79	5.52	10.45	1.54
5M	15.88	5.59	10.45	1.54
10M	16.75	5.90	11.02	1.6

(b) Output Buffer Size (in KB)

Table A.10: Data for Figure 5.3 that shows the performance and area overheads from buffering external outputs in hardware on fault-free execution.

Chkpt	apache	sshd	squid	mysql
10K	40.32	100.00	64.22	76.10
100K	163.15	450.00	176.62	154.51
1M	724.32	1410.48	661.18	977.83
2M	1121.47	5260.68	1268.71	1345.39
5M	4042.58	5921.50	1428.08	4074.19
10M	4440.44	6504.26	3749.69	5472.35

Table A.11: Data for Figure 5.4 that shows (in KB) the sizes of the memory logs for various checkpoint intervals.

Chkpt	Masked	Detected + Recovered	Detected + Unrecovered	Potential SDC
10K	317	7521	833	267
100K	338	7834	693	82
1M	614	7638	643	40
10M	1919	6704	300	36

(a) Permanent Faults

Chkpt	Masked	Detected + Recovered	Detected + Unrecovered	Potential SDC
10K	7731	899	230	70
100K	7724	913	187	46
1M	7750	990	176	39
10M	7858	911	120	35

(b) Transient Faults

Table A.12: Data for Figure 5.5 that shows the outcome of detecting and recovering from permanent and transient faults injected into the server workloads.

Chkpt	System	Masked	Detected + Recovered	Detected + Unrecovered	Potential SDC
10K	Full	317	7621	733	267
	No Dev	317	5938	2413	271
	No I/O	318	5525	2848	265
100K	Full	338	7934	593	82
	No Dev	336	4885	3644	82
	No I/O	335	4467	4083	70
1M	Full	614	7738	543	40
	No Dev	613	4177	4089	37
	No I/O	342	4160	4418	36
10M	Full	1919	6804	200	36
	No Dev	1919	4292	2715	34
	No I/O	353	4276	4307	20

Table A.13: Data for Figure 5.6 that shows the importance of device recovery and output buffering for system recovery in the presence of permanent faults in I/O intensive server workloads..

Application	Ref Input	Test Input
gcc	1.41	0.43
gzip	0.0	0.72
mcf	0.04	0.01
bzip2	0.63	1.31
twolf	0.14	0.02
parser	2.15	1.33
crafty	0.4	0.17
eon	3.42	5.15
gap	0.97	6.03
perlbmk	0.0	0.38
vortex	0.14	0.22
vpr	1.58	0.31
equake	21.22	5.67
ammp	4.6	26.44
art	0.0	0.00
mesa	2.64	4.68

Table A.14: Data for Figure 6.2 that shows the percentage of data-only values for the SPEC workloads under the `ref` and `test` input sets.

App	1M	2M	10M	20M
gcc	1.47%	0.89%	0.43%	0.35%
gzip	2.43%	2.37%	0.72%	0.03%
mcf	0.43%	0.05%	0.01%	0.01%
bzip2	1.35%	1.31%	1.31%	1.28%
twolf	0.10%	0.06%	0.02%	0.02%
parser	1.50%	1.38%	1.33%	1.32%
crafty	0.25%	0.18%	0.17%	0.17%
eon	5.46%	5.15%	5.15%	5.15%
gap	6.10%	6.03%	6.03%	0.63%
perlbmk	0.84%	0.51%	0.38%	0.38%
vortex	0.26%	0.23%	0.22%	0.22%
vpr	0.74%	0.48%	0.31%	0.24%
equake	5.69%	5.67%	5.67%	5.67%
ammp	26.45%	26.44%	26.44%	26.44%
art	6.19%	0.44%	0.00%	0.00%
mesa	7.98%	4.68%	4.68%	4.68%
Average	4.20%	3.49%	3.30%	2.91%

Table A.15: Data for Figure 6.3 that shows the change in the number of data-only values as the window of propagation is increased from 1M instructions till 20M instructions.

App	Injected	Masked	Detected-Short	Detected-Long	SDC
gcc	3003	1648	0	1350	5
gzip	4004	3784	0	220	0
mcf	255	71	0	54	35
bzip2	4004	2720	0	3	747
twolf	561	462	0	3	96
parser	4004	2809	0	1127	68
crafty	4004	3310	0	188	506
eon	4004	3267	0	15	722
gap	3003	2913	0	89	0
perlbmk	3327	3315	0	12	0
vortex	3385	3218	0	117	50
vpr	3219	3035	0	175	2
equake	4004	3954	0	0	50
ammp	4004	3047	0	6	902
art	0	0	0	0	0
mesa	4004	4004	0	0	0
Avg	48785	41557	0	3359	3183

(a) Faults in Data-Only Values

App	Injected	Masked	Detected-Short	Detected-Long	SDC
gcc	4000	2777	1159	56	8
gzip	4000	2591	1223	122	64
mcf	4000	2429	1265	24	277
bzip2	4000	2504	1138	272	49
twolf	4000	2418	1273	9	299
parser	4000	2989	927	28	52
crafty	4000	2639	1166	23	158
eon	4000	3066	610	4	317
gap	4000	2355	1215	240	181
perlbmk	4000	3925	60	14	0
vortex	4000	2642	1277	38	43
vpr	4000	2307	1586	69	38
equake	4000	2409	1563	11	17
ammp	4000	2532	919	4	497
art	4000	2019	1977	4	0
mesa	4000	3111	885	4	0
Avg	64000	42713	18243	922	2000

(b) Faults in Random Values

Table A.16: Data for Figure 6.4 that shows the outcome of architecture-level transient faults injected into (a) data-only values and (b) random values for the SPEC workloads.

App	Total Coverage	False Positives
gcc	42.07%	24.60%
gzip	0.00%	N/A
mcf	19.57%	16.28%
bzip2	37.77%	21.77%
twolf	74.44%	1.47%
parser	14.73%	78.82%
crafty	59.08%	32.12%
eon	33.26%	58.89%
gap	86.67%	14.29%
perlbmk	0.00%	100.00%
vortex	18.56%	22.50%
vpr	32.95%	6.45%
Avg	34.84%	42.59%

Table A.17: Data for Figure 6.5 that shows the coverage and false positives in detecting hard-to-detect faults in data-only values with invariants on all data-only values in the workloads.

App	1 Detector	10 Detectors	100 Detectors
gcc	6.13%	61.18%	32.69%
gzip	87.73%	12.27%	0.00%
mcf	7.07%	87.50%	5.43%
bzip2	18.89%	81.11%	0.00%
twolf	49.49%	45.46%	5.05%
parser	59.50%	11.46%	29.04%
crafty	25.94%	53.74%	20.32%
eon	6.36%	35.88%	57.76%
gap	11.11%	78.89%	10.00%
perlbmk	8.33%	91.67%	0.00%
vortex	16.77%	51.49%	31.74%
vpr	1.63%	44.02%	54.35%
equake	16.00%	66.00%	18.00%
ammp	0.84%	6.06%	31.24%
art	0.00%	0.00%	0.00%
mesa	0.00%	0.00%	0.00%
Avg	22.62%	40.81%	27.23%

Table A.18: Data for Figure 6.6 that shows the coverage of hard-to-detect faults with oracular detectors placed using the fanout metric.

References

[1] N.R. Adiga et al. An overview of the BlueGene/L Supercomputer. In *Proceedings of the International Symposium on Super Computing*, 2002.

[2] Saurabh Agarwal, Rahul Garg, Meeta S. Gupta, and Jose E. Moreira. Adaptive incremental checkpointing for massively parallel systems. In *Proceedings of the International Symposium on Super Computing*, 2004.

[3] Periklis Akritidis et al. Preventing Memory Error Exploits with WIT. In *Proceedings of the Symposium on Operating Systems Principles*, 2008.

[4] Todd M. Austin. DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design. In *Proceedings of the International Symposium on Microarchitecture*, 1998.

[5] A. Avizienis. The N-Version Approach to Fault-Tolerant Software. *IEEE Transaactions of Software Engineering*, 11(12), 1985.

[6] A. Benso, S. Di Carlo, G. Di Natale, P. Prinetto, and L. Tagliaferri. Data Critically Estimation In Software Applications. In *Proceedings of the International Test Conference*, 2003.

[7] David Bernick et al. NonStop Advanced Architecture. In *Proceedings of the International Conference on Dependable Systems and Networks*, 2005.

[8] Shekhar Borkar. Designing Reliable Systems from Unreliable Components: The Challenges of Transistor Variability and Degradation. *IEEE Micro*, 25(6), 2005.

[9] Fred Bower, Daniel Sorin, and Sule Ozev. Online Diagnosis of Hard Faults in Microprocessors. *Proceedings of the ACM Transactions on Architecture and Code Optimization*, 4(2), 2007.

[10] Greg Bronevetsky, Daniel Marques, Keshav Pingali, and Radu Rugina. Compiler-Enhanced Incremental Checkpointing. *20th Intl. Workshop on Languages and Compilers for Parallel Computing*, 2008.

[11] Jonathan Chang, George Reis, and David August. Automatic Instruction-Level Software-Only Recovery. In *Proceedings of the International Conference on Dependable Systems and Networks*, 2006.

[12] Kypros Constantinides et al. Software-Based On-Line Detection of Hardware Defects: Mechanisms, Architectural Support, and Evaluation. In *Proceedings of the International Symposium on Microarchitecture*, 2007.

[13] Jonathan Corbet and Greg Kroah-Hartman Alessandro Rubini. *Linux Device Drivers*. O'Reilly, 3rd edition, 2005.

[14] Oracle Corporation. Solaris Device Driver Tutorial, 2010.

[15] cURL. Tool for transfering files with URL syntax. Website. `http://curl.haxx.se`.

[16] Marc de Kruijf, Shuou Nomura, and Karthikeyan Sankaralingam. Relax: An Architectural Framework for Software Recovery of Hardware Faults. In *Proceedings of the International Symposium on Computer Architecture*, 2010.

[17] Marc de Kruijf and Karthikeyan Sankaralingam. Exploring the Synergy of Emerging Workloads and Silicon Reliability Trends. In *Proceedings of the Workshop on Silicon Errors in Logic – System Effects*, 2009.

[18] A. DeHorn, N. Carter, and H. Quinn. Final Report on CCC Cross-Layer Reliability Visioning Study, 2011.

[19] Joe Devietti, Colin Blundell, Milo Martin, and Steve Zdancewic. Hardbound: Architectural Support for Spatial Safety of the C Programming Language. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2008.

[20] Dinakar Dhurjati, Sumant Kowshik, and Vikram Adve. SAFECode: Enforcing Alias Analysis for Weakly Typed Languages. *SIGPLAN Not.*, 41(6), 2006.

[21] Martin Dimitrov and Huiyang Zhou. Unified Architectural Support for Soft-Error Protection or Software Bug Detection. In *Proceedings of the International Conference on Parallel Archtectures and Compilation Techniques*, 2007.

[22] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. In *Proceedings of the International Symposium on Operating Systems Design and Implmentation*, 2002.

[23] Dan Ernst et al. Razor: A Low-Power Pipeline Based on Circuit-Level Timing Speculation. In *MICRO*, dec 2003.

[24] Michael D. Ernst et al. The Daikon System for Dynamic Detection of Likely Invariants. *Science of Computer Programming*, 2007.

[25] Roberto Gioiosa, Jose Carlos Sancho, Song Jiang, and Fabrizio Petrini. Transparent, Incremental Checkpointing at Kernel Level: A Foundation for Fault Tolerance for Parallel Computers. In *Proceedings of the International Symposium on Super Computing*, 2005.

[26] O. Goloubeva et al. Soft-Error Detection Using Control Flow Assertions. In *Proceedings of the International Conference on Digital Fault Testing*, 2003.

[27] Greg Hamerly, Erez Perelman, Jeremy Lau, and Brad Calder. SimPoint 3.0: Faster and More Flexible Program Analysis. In *Workshop on Modeling, Benchmarking and Simulation*, June 2005.

[28] Siva Hari, Manlap Li, Pradeep Ramachandran, and Sarita Adve. Low-cost Hardware Fault Detection and Diagnosis for Multicore Systems. In *Proceedings of the International Symposium on Microarchitecture*, 2009.

[29] Siva Kumar Sastry Hari, Helia Naeimi, Pradeep Ramachandran, and Sarita Adve. Relyzer: Application Resiliency Ananlyzer for Transient Faults. In *Proceedings of the Workshop on Silicon Errors in Logic – System Effects*, 2011.

[30] Asim Kadav et al. Tolerating Hardware Device Failures in Software. In *Proceedings of the Symposium on Operating Systems Principles*, 2009.

[31] G. Kanawati et al. FERRARI: A Flexible Software-based Fault and Error Injection System. *IEEE Computer*, 44(2), 1995.

[32] Andrew Lenharth, Vikram S. Adve, and Samuel T. King. Recovery Domains: An Organizing Principle for Recoverable Operating Systems. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2009.

[33] Manlap Li, Pradeep Ramachandran, Rahmet Ulya Karpuzcu, Siva Hari, and Sarita Adve. Accurate Microarchitecture-Level Fault Modeling for Studying Hardware Faults. In *Proceedings of the International Symposium on High Performance Computer Architecture*, 2009.

[34] Manlap Li, Pradeep Ramachandran, Swarup Sahoo, Sarita Adve, Vikram Adve, and Yuanyuan Zhou. Trace-Based Microarchitecture-Level Diagnosis of Permanent Hardware Faults. In *Proceedings of the International Conference on Dependable Systems and Networks*, 2008.

[35] Manlap Li, Pradeep Ramachandran, Swarup Sahoo, Sarita Adve, Vikram Adve, and Yuanyuan Zhou. Understanding the Propagation of Hard Errors to Software and Implications for Resilient Systems Design. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2008.

[36] Xuanhua Li and Donald Yeung. Application-level correctness and its impact on fault tolerance. In *Proceedings of the International Symposium on High Performance Computer Architecture*, 2007.

[37] Jien-Chung Lo. Reliable Floating-Point Arithmetic Algorithms for Error-Coded Operands. *IEEE Transactions on Computers*, 43, 1994.

[38] M. J. Mack, W. M. Sauer, S. B. Swaney, and B. G. Mealey. IBM POWER6 reliability. *IBM Journal of Research and Development*, 51(6), November 2007.

[39] Peter B. Mark. The Sequoia Computer: A Fault-Tolerant Tightly-Coupled Multiprocessor Architecture. In *Proceedings of the International Symposium on Computer Architecture*, 1985.

[40] Milo Martin et al. Multifacet's General Execution-Driven Multiprocessor Simulator (GEMS) Toolset. *SIGARCH Computer Architecture News*, 33(4), 2005.

[41] Yoshio Masubuchi, Satoshi Hoshina, Tomofumi Shimada, Hideaki Hirayama, and Nobuhiro Kato. Fault Recovery Mechanism for Multiprocessor Servers. In *Proceedings of the International Symposium on Fault-Tolerant Computing*, 1997.

[42] Carl Mauer, Mark Hill, and David Wood. Full-System Timing-First Simulation. *SIGMETRICS Perf. Eval. Rev.*, 30(1), 2002.

[43] Richard McDougall and Jim Mauro. *Solaris Internals: Solaris 10 and OpenSolaris Kernel Architecture*. Prentice Hall, 2nd edition, 2006.

[44] Albert Meixner, Michael E. Bauer, and Daniel Sorin. Argus: Low-Cost, Comprehensive Error Detection in Simple Cores. In *Proceedings of the International Symposium on Microarchitecture*, 2007.

[45] Gordon E. Moore. Cramming more Components onto Integrated Circuits. *Electronics*, 38(8), April 1965.

[46] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. LogTM: Log-Based Transactional Memory. In *Proceedings of the International Symposium on High Performance Computer Architecture*, 2006.

[47] M. Mueller et al. RAS Strategy for IBM S/390 G5 and G6. *IBM Journal on Research and Development*, 43(5/6), Sept/Nov 1999.

[48] Shubhendu Mukherjee et al. The Soft Error Problem: An Architectural Perspective. In *Proceedings of the International Symposium on High Performance Computer Architecture*, 2005.

[49] Onur Mutlu, Jared Stark, Chris Wilkerson, and Yale Patt. Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-order Processors. In *Proceedings of the International Symposium on High Performance Computer Architecture*, 2003.

[50] Santosh Nagarakatte, Jianzhou Zhao, Milo Martin, and Steve Zdancewic. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In *Proceedings of the International Conference on Programming Language Design and Implementation*, 2009.

[51] Jun Nakano et al. ReVive I/O: Efficient Handling of I/O in Highly-Available Rollback-Recovery Servers. In *Proceedings of the International Symposium on High Performance Computer Architecture*, 2006.

[52] Nithin Nakka et al. An Architectural Framework for Detecting Process Hangs/Crashes. In *European Dependable Computing Conf*, 2005.

[53] Jakob Nielsen. *Usability Engineering*. Morgran Kaufmann, 1st edition, 1993.

[54] Praveen Parvathala, Kaila Maneparambil, and William Lindsay. FRITS: A Microprocessor Functional BIST Method. In *Proceedings of the International Test Conference*, 2002.

[55] Karthik Pattabiraman et al. Dynamic Derivation of Application-Specific Error Detectors and their Implementation in Hardware. In *European Dependable Computing Conference*, 2006.

[56] Karthik Pattabiraman, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Application-based metrics for strategic placement of detectors. In *Proceedings of the Pacific Rim International Symposium on Dependable Computing*, 2005.

[57] Andrea Pellegrini et al. CrashTest: A Fast High-Fidelity FPGA-based Resiliency Analysis Framework. In *Proceedings of the International Conference on Computer Design*, 2008.

[58] Andrea Pellegrini, Robert Smolinski, Lei Chen, Xin Fu, Siva Kumar Sastry Hari, Junhao Jiang, Sarita Adve, Todd Austin, and Valeria Bertacco. CrashTest'ing SWAT: Accurate, Gate-Level Evaluation of Symptom-Based Resiliency Solutions. In *Proceedings of the Workshop on Silicon Errors in Logic – System Effects*, 2011.

[59] Michael D. Powell, Arijit Biswas, Shantanu Gupta, and Shubhendu S. Mukherjee. Architectural core salvaging in a multi-core processor for hard-error tolerance. In *Proceedings of the International Symposium on Computer Architecture*, 2009.

[60] Milos Prvulovic et al. ReVive: Cost-Effective Arch Support for Rollback Recovery in Shared-Mem Multiprocessors. In *Proceedings of the International Symposium on Computer Architecture*, 2002.

[61] Paul Racunas et al. Perturbation-based Fault Screening. In *Proceedings of the International Symposium on High Performance Computer Architecture*, 2007.

[62] L. Rashid, K. Pattabiraman, and S. Gopalakrishnan. Towards Understanding the Effects of Intermittent Hardware Faults on Programs. In *Proceedings of the International Conference on Dependable Systems and Networks*, 2010.

[63] V. Reddy et al. Assertion-Based Microarchitecture Design for Improved Fault Tolerance. In *Proceedings of the International Conference on Computer Design*, 2006.

[64] George Reis et al. Software-Controlled Fault Tolerance. *Proceedings of the ACM Transactions on Architecture and Code Optimization*, 2(4), 2005.

[65] Bogdan Romanescu and Daniel Sorin. Core Cannibalization Architecture: Improving Lifetime Chip Performance for Multicore Processors in the Presence of Hard Faults. In *Proceedings of the International Conference on Parallel Archtectures and Compilation Techniques*, 2008.

[66] Swarup Sahoo et al. Using Likely Program Invariants to Detect Hardware Errors. In *Proceedings of the International Conference on Dependable Systems and Networks*, 2008.

[67] Design Panel, SELSE II - Reverie, 2006. `http://www.selse.org/selse2.org/recap.pdf`.

[68] Smitha Shyam et al. Ultra Low-Cost Defect Protection for Microprocessor Pipelines. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.

[69] Stelios Sidiroglou, Oren Laadan, Carlos Perez, Nicolas Viennot, Jason Nieh, and Angelos D. Keromytis. ASSURE: Automatic Software Self-Healing Using Rescue Points. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2009.

[70] Jared C. Smolens, Brian T. Gold, Jangwoo Kim, Babak Falsafi, James C. Hoe, and Andreas G. Nowatzyk. Fingerprinting: Bounding Soft-Error Detection Latency and Bandwidth. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2004.

[71] Daniel Sorin et al. SafetyNet: Improving the Availability of Shared Memory Multiprocessors with Global Checkpoint/Recovery. In *Proceedings of the International Symposium on Computer Architecture*, 2002.

[72] 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.

[73] V. Sridharan and D.R. Kaeli. Eliminating Microarchitectural Dependency from Architectural Vulnerability. In *Proceedings of the International Symposium on High Performance Computer Architecture*, 2009.

[74] Micael Swift, Muthukaruppan Annamalai, Brian Bershad, and Henry Levy. Recovering Device Drivers. In *Proceedings of the International Symposium on Operating Systems Design and Implmentation*, 2004.

[75] Rajesh Venkatasubramanian et al. Low-Cost On-Line Fault Detection Using Control Flow Assertions. In *Proceedings of the International Online Test Symposium*, 2003.

[76] Virtutech. Simics Full System Simulator. Website, 2006. `http://www.simics.net`.

[77] Nicholas Wang, Michael Fertig, and Sanjay Patel. Y-Branches: When You Come to a Fork in the Road, Take It. In *Proceedings of the International Conference on Parallel Archtectures and Compilation Techniques*, 2003.

[78] N.J. Wang and S.J. Patel. ReStore: Symptom-Based Soft Error Detection in Microprocessors. *IEEE Transactions on Dependable and Secure Computing*, 3(3), July-Sept 2006.

[79] David Yen. Chip Multithreading Processors Enable Reliable High Throughput Computing. In *Proceedings of the International Reliability Physics Symposium*, 2005. Keynote Address.

[80] Keun Soo Yim and Ravishankar Iyer. Hauberk: Lightweight Silent Data Corruption Error Detectors for GPGPU. In *In Proceedings of the 17th Humantech Thesis Prize (Also in IPDPS 2011)*, 2011.

[81] 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 *Proceedings of the International Symposium on Microarchitecture*, 2004.

[82] Pin Zhou, Feng Qin, Wei Liu, Yuanyuan Zhou, and Josep Torrellas. iWatcher: Simple, General Architectural Support for Software Debugging. In *Micro Special Issue: Micro's Top Picks from Computer Architecture Conferences*, 2004.

Author's Biography

Pradeep Ramachandran was born in Madras, India on the 2^{nd} of March, 1984. He finished his schooling at St. John's English School and Junior College, Besant Nagar and graduated high school from P. S. Senior Secondary School, Mylapore. He received his Bachelor of Technology degree in Computer Science and Engineering from the Indian Institute of Technology (IIT), Madras in 2005 and his Master of Science degree from the University of Illinois at Urbana Champaign in 2007. He was awarded an IBM PhD scholarship and an Intel PhD fellowship in 2009, and the W.J.Poppelbaum Memorial Award in 2011.