Integrated Hardware-Software Diagnosis of Intermittent Faults

by

Majid Dadashikelayeh

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF APPLIED SCIENCE

in

The Faculty of Graduate and Postdoctoral Studies

(Electrical and Computer Engineering)

THE UNIVERSITY OF BRITISH COLUMBIA

(Vancouver)

August 2014

Abstract

Intermittent hardware faults are hard to diagnose as they occur non-deterministically. Hardware-only diagnosis techniques incur significant power and area overheads. On the other hand, software-only diagnosis techniques have low power and area overheads, but have limited visibility into many micro-architectural structures and hence cannot diagnose faults in them.

To overcome these limitations, we propose a hardware-software integrated framework for diagnosing intermittent faults. The hardware part of our framework, called SCRIBE continuously records the resource usage information of every instruction in the processor, and exposes it to the software layer. SCRIBE has 0.95% on-chip area overhead, incurs a performance overhead of 12% and power overhead of 9%, on average. The software part of our framework is called SIED and uses backtracking from the program's crash dump to find the faulty micro-architectural resource. Our technique has an average accuracy of 84% in diagnosing the faulty resource, which in turn enables fine-grained deconfiguration with less than 2% performance loss after deconfiguration.

Preface

- Parts of Chapters 1, 2, 3 and 5 have been appeared in the IEEE Silicon Errors in Logic - System Effects 2013 workshop.

  Majid Dadashi, Layali Rashid and Karthik Pattabiraman, SCRIBE: A Hardware Infrastructure Enabling Fine-Grained Software Layer Diagnosis. Poster presentation at the 9th Workshop on Silicon Errors in Logic - System Effects (SELSE), 2013.

- Parts of Chapters 1, 1.3, 2, 3, 4, 5 and 1.4 have been published in the IEEE/IFIP International Conference on Dependable Systems and Networks 2014. I setup all the experiments and wrote most of the paper.

  Majid Dadashi, Layali Rashid, Karthik Pattabiraman, and Sathish Gopalakrishnan. 2014. Integrated Hardware-Software Diagnosis of Intermittent Faults .DSN, 2014.

Table of Contents

List of Tables

List of Figures

Acknowledgements

First and foremost, I would like to express my deepest appreciation to my advisor Dr. Karthik Pattabiraman. His patience and diligence when faced with problems, his passion for research and his constant support have been paramount in making this thesis possible.

I would also like to thank the other co-authors of my papers, Dr. Sathish Gopalakrishnan and Dr. Layali Rashid whose work has been the cornerstone of my research. They have also been truly supportive and provided precious feedback during this work.

I would also like to thank my labmates and colleagues at the Computer Systems Reading Group (CSRG) for many insightful discussions on my research topic as well as on related fields.

Finally, I would like to thank my family for all the support they have given me throughout the years. I would definitely not be here without them.

Dedication

To my family and friends

Chapter 1

Introduction

1.1 Motivation

CMOS scaling has exacerbated the unreliability of Silicon devices and made them more susceptible to different kinds of faults [4]. The common kinds of hardware faults are transient and permanent. However, a third category of faults, namely intermittent faults has gained prominence [11]. A recent study of commodity hardware has found that intermittent faults were responsible for at least 39% of computer system failures due to hardware errors [22]. Unlike transient faults, intermittent faults are not one-off events, and occur repeatedly at the same location. However, unlike permanent faults, they appear non-deterministically, and only in certain circumstances.

Diagnosis is an essential operation for a fault-tolerant system. In this thesis, we focus on diagnosing intermittent faults that occur in the processor. Intermittent faults are caused by marginal or faulty micro-architectural components, and hence diagnosing such faults is important to isolate the faulty resource [6, 16, 33]. Components can experience intermittent faults either due to design and manufacturing errors, or due to aging and temperature effects that arise in operational settings [11]. Therefore, the diagnosis process should be run throughout the life-time of the processor rather than

only at design validation time. This makes it imperative to design a diagnosis scheme that has low online performance and power overheads. Further, to retain high performance after repair, the diagnosis should be fine-grained at the granularity of individual resources in a micro-processor, so that the processor can be deconfigured around the faulty resource after diagnosis [25].

Diagnosis can be carried out in either hardware or software. Hardware-level diagnosis has the advantage that it can be done without software changes. Unfortunately, performing diagnosis entirely in hardware incurs significant power and area overheads, as diagnosis algorithms are often complex and require specialized hardware to implement. On the other hand, software-based diagnosis techniques only incur power and performance overheads during the diagnosis process, and have zero area overheads. Unfortunately, software techniques have limited visibility into many micro-architectural structures (e.g., the reorder buffer) and hence cannot diagnose faults in them. Further, software techniques cannot identify the resources consumed by an instruction as it moves through the pipeline, which is essential for fine-grained diagnosis.

1.2 Proposed Solution and Contributions

In this thesis, we propose a hardware-software integrated technique for diagnosing intermittent hardware errors in multi-core processors. As mentioned above, intermittent faults are non-deterministic and may not be easily reproduced through posteriori testing. Therefore, the hardware portion of our technique continuously records the micro-architectural resources used by an

instruction as the instruction moves through the processor's pipeline, and stores this information in a log that is exposed to the software portion of the technique. We call the hardware portion SCRIBE. When the program fails (due to an intermittent fault), the software portion of our technique uses the log to identify which resource of the microprocessor was subject to the intermittent fault that caused the program to fail. The software portion runs on a separate core and uses a combination of deterministic replay and back-tracking from the failure point, to identify the faulty component. We call the software portion of our technique SIED, which stands for *Software-based Intermittent Error Diagnosis*. SCRIBE and SIED work in tandem to achieve intermittent fault diagnosis.

Prior work on diagnosis [6] has either assumed the presence of fine-grained checkers such as the DIVA checker [2], or has assumed that the fault occurs deterministically [19], which is true for permanent faults, but not intermittent faults. In contrast, our technique does not require any fine-grained checkers in the processor nor does it rely upon determinism of the fault, making it well suited for intermittent faults. Several papers have proposed diagnosis mechanisms for post-Silicon validation [8, 23]. However, these approaches target design faults and not operational faults, which is our focus. *To the best of our knowledge, we are the first to propose a general purpose diagnosis mechanism for in-field, intermittent faults in processors, with minimal changes to the hardware.*

The main contributions of this thesis are as follows:

1. Enumerate the challenges associated with intermittent fault diagno-

sis and explain why a hybrid hardware-software scheme is needed for diagnosis.

2. Propose SCRIBE, an efficient micro-architectural mechanism to record instruction information as it moves through the pipeline, and expose this information to the software layer.

3. Propose SIED, a software-based diagnosis algorithm that leverages the information provided by SCRIBE to isolate the faulty micro-architectural resource through backtracking from the failure point,

4. Conduct an end-to-end evaluation of the hybrid approach in terms of diagnosis accuracy using fault injection experiments at the micro-architectural level.

5. Evaluate the performance, power and on-chip area overheads incurred by SCRIBE during fault-free operation. Also, evaluate the overhead incurred by the processor after it is deconfigured upon a successful diagnosis by our approach.

Our experiments on the SPEC2006 benchmarks show that SCRIBE incurs an average performance overhead of 11.5%, a power consumption overhead of 9.3% and on-chip area overhead of 0.95% for a medium-width processor. Further, the end-to-end accuracy of diagnosis is 84% on average across different resources of the processor (varies from 71% to 95% depending on the pipeline stage in which the fault occurs). We also show that with such fine-grained diagnosis, only 1.6% performance overhead will be incurred by the processor after deconfiguration, on average.

1.3 Background

In this section, we first explain what are intermittent faults, and their causes. We then explain why resource level, online diagnosis is needed for multi-core processors. Finally, we explain the Dynamic Dependence Graph (DDG), which is used in this thesis for diagnosis.

1.3.1 Intermittent Faults: Definition and Causes

Definition We define an intermittent fault as one that appears non- deterministically at the same hardware location, and lasts for one or more (but finite number of) clock cycles. The main characteristic of intermittent faults that distinguishes them from transient faults is that they occur repeatedly at the same location, and are caused by an underlying hardware defect rather than a one-time event such as a particle strike. However, unlike permanent faults, intermittent faults appear non-deterministically, and only under certain conditions.

Causes: The major cause of intermittent faults is device wearout, or the tendency of solid-state devices to degrade with time and stress. Wearout can be accelerated by aggressive transistor scaling which makes processors more susceptible to extreme operating condition such as voltage and temperature fluctuations [21], [5]. In-progress wearout faults are often intermittent as they depend on the operating conditions and the circuit inputs. In the long term, such faults may eventually lead to permanent defects. Another cause of intermittent faults is manufacturing defects that escape VLSI testing [10]. Often, deterministic defects are flushed out during such testing and the

ones that escape are non-deterministic defects, which emerge as intermittent faults. Finally, design defects can also lead to intermittent faults, especially if the defect is triggered under rare scenarios or conditions [32]. However, we do not consider intermittent faults due to design defects in this thesis.

1.3.2 Why Resource-Level, Online Diagnosis ?

Our goal is to isolate individual microarchitectural resources and units that are responsible for the intermittent fault. Fine-grained diagnosis implicitly assumes that these resources can be deconfigured dynamically in order to prevent the fault from occurring again. Other work has also made similar assumptions [6], [19], [16]. While it may be desirable to go even further and isolate individual circuits or even transistors that are faulty, it is often difficult to perform deconfiguration at that level. Therefore, we confine ourselves to performing diagnosis at the resource level.

Another question that arises in fine-grained diagnosis is why not simply avoid using the faulty core instead of deconfiguring the faulty resource. This would be a simple and cost-effective solution. However, this leads to vastly lower performance in a high-performance multi-core processor, as prior work has shown [25], [16]. Finally, the need for online diagnosis stems from the fact that taking the entire processor or chip offline to perform diagnosis is wasteful, especially as the rate of intermittent faults increases as future trends indicate [10]. Further, taking the chip offline is not feasible for safety-critical systems. Our goal is to perform online diagnosis of intermittent faults.

1.3.3 Dynamic Dependency Graphs

A dynamic dependency graph (DDG) is a representation of data flow in a program [1]. It is a directed acyclic graph where graph nodes or vertices represent values produced by dynamic instructions during program execution. In effect, each node corresponds to a dynamic instance of a value-producing program instruction. Dependencies among nodes result in edges in the DDG. In the DDG, there is an edge from node N_1 (corresponding to instruction I_1) to node N_2 (corresponding to instruction I_2), if and only if I_2 reads the value written by I_1 (instructions that do not produce any values correspond to nodes with no outgoing edges).

1.4 Related Work

Bower et al. [6] propose a hardware-only diagnosis mechanism by modifying the processor pipeline to track the resources used by an instruction (similar to SCRIBE), and finding the faulty resources based on resource counters. However, their scheme relies on the presence of a fine-grained checker (e.g., DIVA [2]) to detect errors before an instruction commits. This limits its applicability to processors that are specifically designed with such fine-grained checkers.

Li et al. [19] use a combination of hardware and software to diagnose permanent errors. Similar to our approach, theirs is also a hybrid technique that splits the diagnosis between hardware and software. However, they rely on the determinism of the fault, as they replay a failed program execution (due to a permanent fault) from a checkpoint and gather its micro-architectural

resource usage information during the replay. Unfortunately, this technique would not work for intermittent faults that are non-deterministic, as the fault may not show up during the replay.

IFRA [23], is a post-silicon bug localization method, which records the footprint of every instruction as it is executed in the processor. IFRA is similar to SCRIBE in how it records the information. However, SCRIBE differs from IFRA in two ways. First, IFRA records the instruction information within the processor, and this information is scanned out after the failure, after the processor is stopped. On the other hand, SCRIBE writes the gathered information to memory during regular operation. Second, IFRA required the presence of hardware-based fault detectors to limit the error propagation. In contrast, SCRIBE does not require any additional detectors in the hardware or software.

DeOrio et al. [13] introduce a hybrid hardware-software scheme for post-silicon debugging mechanism, in which the hardware logs the signal activities during post-silicon validation, and the software uses anomaly detection on the logged signals to identify a set of candidate root-cause signals for a bug. Because their focus is on post-silicon debugging, they do not present the performance overhead of their technique, and hence it is not possible for us to compare their performance overheads with ours.

Carratero et al. [8] performs integrated hardware-software diagnosis for faults in the Load-Store Unit (LSU). Our work is similar to theirs in some respects. However, our approach covers faults in the entire pipeline, and not only the Load Store Unit. Further, their goal is to diagnose design faults during post-silicon validation, while ours is to diagnose intermittent faults

during regular operation.

There has been considerable work on online testing for fault diagnosis. For example, Constantinides et al. [12] propose a periodic mechanism to run directed tests on the hardware using a dedicated set of instructions. However, this technique may find errors that do not affect the application, which in turn may initiate unnecessary recovery or repair actions, thus resulting in high overheads. To mitigate this problem, Pellegrini and Bertacco. [24] propose a hybrid hardware-software solution that monitors the hardware resource usage in the application, and tests only the resources that are used by the application. While this is useful, all testing-based methods require that the fault appears during at least one of the testing phases, which may not hold for intermittent faults.

1.5 Summary

In this chapter, we explain the motivations behind the online fine-grained diagnosis of hardware intermittent faults. Hardware intermittent faults are becoming more prominent with the technology scaling. We perform diagnosis with the granularity of micro-architectural units. Finding the root cause of an intermittent fault in this level enables the automatic repair of the chip using deconfiguration with minimal performance loss after repair.

In the following chapters, we present a hybrid hardware/software mechanism for diagnosing hardware intermittent faults. We evaluate different overheads of our design using simulation. We also perform fault injection experiments to evaluate the accuracy of our diagnosis scheme.

Chapter 2

Approach

This section first presents the fault model we consider. It then presents the challenges of intermittent fault diagnosis. Finally, it presents an overview of our approach and how it addresses the challenges.

2.1 Fault Model

As mentioned in Section 1.3.1, intermittent faults are faults that last for finite number of cycles at the same micro-architectural location. We consider intermittent faults that occur in processors. In particular, we consider faults that occur in *functional units*, *reorder buffer*, *instruction fetch queue*, *load/store queue* and *reservation station entries*. We assume that caches and register files are protected using ECC or parity and therefore do not experience software visible faults. We also assume that the processor's control logic is immune to errors, as this is a relatively small portion of the chip [27]. Finally, we assume that a component may be affected by at most one intermittent fault at any time, and that the fault affects a single bit in the component (stuck-at zero/one), lasting for several cycles.

2.2 Challenges

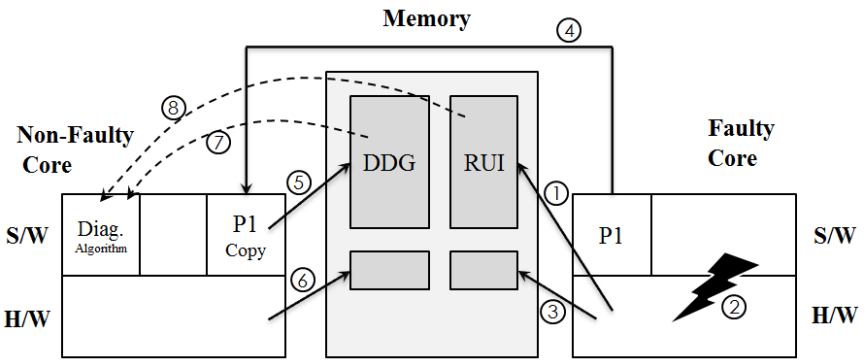
In this section, we outline the challenges that an intermittent fault diagnosis method needs to overcome.

Non-determinism: Since intermittent faults occur non-deterministically, re-execution of a program that has failed as a result of an intermittent fault, often results in a different event sequence than the original execution. In other words, the sequence of events that lead to a failure is not (necessarily) repeatable under intermittent faults.

Overheads: An intermittent fault diagnosis mechanism should incur as low overhead as possible in terms of performance, area and power, especially during fault-free operation, which is likely to be the common case.

Software Layer Visibility: Software diagnosis algorithms suffer from limited visibility into the hardware layer. In other words, software-only approaches are not aware of what resources an instruction has used since being fetched until retiring from the pipeline (the only inferable information from an instruction is the *type* of functional unit it has used).

No information about the faulty instructions: To find the faulty resource, the diagnosis algorithm needs to have information about the instructions that have been affected by the intermittent fault in order that the search domain of the faulty resource can be narrowed down to resources used by these instructions. One way to obtain this information is to log the value of the destination of every instruction at runtime, and to compare its value with that of a fault-free run (more details in Section 2.3). However, logging the value of every executed instruction in addition to its resource



1. Gather RUI and log to memory (SCRIBE)
2. Failure due to intermittent fault
3. Log program's register and memory state (core dump)
4. Deterministic replay on another core (SIED)
5. Construct replayed program's DDG (SIED)
6. Log replayed program's register and memory state (SIED)
7. Construct augmented DDG and backtrack using analysis heuristics (SIED)

Figure 2.1: End to end scenario of failure diagnosis by SCRIBE and SIED. The steps in the figure are explained in the box.

information can result in prohibitive performance overheads as we show in Section 5. Therefore, we need to infer this information from the failure log instead.

2.3 Overview of Our Approach

In this section, we present an overview of our approach and how it addresses the challenges in section 2.2.

We propose a hybrid hardware-software approach for diagnosis of intermittent faults in processors. Our approach consists of two parts. First, we propose a simple, low-overhead, hardware mechanism called SCRIBE to record information about resource usage of each instruction and expose this information to the software. Second, we propose a software technique called SIED that uses the recorded information upon a failure (caused by an intermittent fault) to diagnose the faulty resource by backtracking from the point of failure through the program's DDG (see Section 1.3.3). The intuition is that errors propagate along the DDG edges starting from the instruction that used the faulty resource, and hence backtracking on the DDG can diagnose the fault.

Assumptions: We make the following assumptions about the system:

i) We assume a commodity multi-core system in which all cores are homogeneous, and are able to communicate with each other through shared address space.

ii) We assume the availability of a fault-free core to perform the diagnosis, e.g. using Dual Modular Redundancy (DMR). This is similar to the assumption made by Li et al. [19]. The fault-free core is only needed during diagnosis.

iii) The processor is able to deterministically replay the failed program's execution. Researchers have proposed the use of deterministic replay techniques for debugging programs on multi-core machines [14, 34]. This is needed to eliminate the effect of non-deterministic events in the program during diagnosis (other than the fault).

iv) The fault has already been identified as an intermittent fault prior

to diagnosis. In particular, it has been ruled out to be a transient fault - this can be done by only invoking diagnosis if there are repeated failures. For example, there has been work on distinguishing intermittent faults from transient faults using a threshold mechanism [3].

Steps : Figure 2.1 shows the sequence of steps our technique would follow to diagnose a fault.

1. As the program executes, the hardware layer SCRIBE logs the Resource Usage Information (RUI) of the instructions (step 1 in Figure 2.1) to memory. Every instruction has an RUI, a bit array indicating which resources it has used while moving through the processor's pipeline. SCRIBE is presented in Section 3.

2. Assume that the program fails as a result of an intermittent fault burst in one of the processor resources (step 2). This failure can occur due to a crash or an error detection by the application (e.g. an assertion failure). The registers and memory state of the application is dumped to memory, typically as a core dump (step 3).

3. The software layer diagnosis process, SIED is started on *another* core. This core is used to perform the diagnosis and is assumed to itself be fault-free during diagnosis (see assumptions). SIED replays the program using deterministic replay mechanisms, and constructs the DDG (steps 4 and 5) of the replayed program. The original program can be resumed on the core that experienced the intermittent fault, as SIED does not interfere with it.

4. When the replayed program reaches the instruction at which the orignal program failed, SIED dumps its register and memory state to memory (step 6).

5. SIED merges the DDG from step 5 with the RUI log in step 1, to build the *augmented DDG*. This is a DDG in which every node contains the RUI of its corresponding instruction in the program.

6. SIED then compares the memory and register states dumped in steps 3 and 6 to identify the set of nodes in the augmented DDG that differ between the original and replayed execution. Because the replayed execution used deterministic replay, any differences between the executions are due to the intermittent fault.

7. Finally, SIED backtracks from the faulty nodes in the augmented DDG using *analysis heuristics* to find the faulty resource (steps 7 and 8). The details of how SIED works are explained in Section 4.

Challenges Addressed: We now illustrate how our technique satisfies the constraints posed in Section 2.2.

Non-determinism: Our technique gathers the micro architectural resource usage information online using the SCRIBE layer (Step 1). Therefore, it requires determinism neither in resource usage nor fault occurrence during the replay.

Overheads: Our technique initiates diagnosis only when a crash or error detection occurs, thus the diagnosis overhead is not incurred during fault-free execution. However, the SCRIBE layer incurs both performance and power

overheads as it continuously logs the instructions' resource usage information executing in the processor. Note that SCRIBE only exposes the hardware RUI information to the software layer. The complex task of figuring out the faulty component is done in software. Hence, the power overhead of SCRIBE is low. We describe the optimizations made to SCRIBE to keep its performance overhead low in Section 3. We present the performance and power overheads in section 5.2.

Software-layer visibility: The SCRIBE layer records the information on micro-architectural resource usage and exposes it to software, thus solving the visibility problem.

No information about faulty instructions: Our technique does not log the destination result of each instruction, and hence cannot tell which instructions have been affected by the fault. Instead, SIED uses the replay run to determine which registers/memory locations are affected by the fault, and backtracks from these in the DDG to identify the faulty resource.

2.4 Summary

In this chapter, we explain our fault model and a high level overview of our hybrid hardware/software diagnosis scheme. Our fault model is stuck-at-0 and stuck-at-1 with random occurrence cycle and random duration. The fault can occur in any of the entries of the *Reorder Buffer*, *Reservation Station*, *Instruction Fetch Queue* and *Load Store Queue*. The fault could also occur at any of the individual *functional units*.

The diagnosis scheme consists of a hardware layer called *SCRIBE* and a

software layer called *SIED*. SCRIBE is responsible for providing the information about the resource usage of the instructions when being executed in the pipeline. SIED performs a deterministic replay of the process to extract its dynamic dependence graph. It then augments the dynamic dependence graph with the resource usage information provided by SCRIBE. In the final stage, SIED performs two DDG analysis algorithms on the augmented DDG to find the faulty unit.

In the next two chapters, we present a detailed explanation of the hardware layer and the software layer.

Chapter 3

SCRIBE: Hardware Layer

We propose a hybrid diagnosis approach involving both hardware and software. SCRIBE is the hardware part of our hybrid scheme and is responsible for exposing the micro-architectural **R**esource **U**sage **I**nformation (RUI) to the software layer, SIED. This allows SIED to identify the faulty resource(s) upon a failure due to an intermittent fault. In addition, SCRIBE also logs the addresses of the executed branches, so that the program's control flow can be restored in case of a failure (Section 4.1). This chapter presents the functionality and the design of SCRIBE.

First we explain format of the data which gets revealed to the software layer. Then the RUI collection mechanism from the pipeline is explained. In the next section, the mechanism and the hardware components for sending data to memory hierarchy is explained and at the end, we explain the *priority handling* technique which decreases the performance overhead to a significant degree.

3.1 RUI Format

We use the term RUI to denote the log corresponding to a single instruction's execution as it moves through the pipeline. The RUI records the units used

by the instruction in each pipeline stage. Each field of the RUI corresponds to a single pipeline stage or functional unit. As an example, we consider an *add* instruction which is put in the entry 4 of Instruction Fetch Queue, entry 7 of the ROB, entry 24 of reservation station and also uses the second integer ALU. The RUI of this instruction is shown in figure 3.1.

The RUIs are stored in a circular buffer in the process's memory address space as the program executes on the processor. The size of the RUI buffer is determined by the worst-case number of instructions taken by programs to crash or fail after an intermittent fault. Because this number can be large, keeping the buffer on chip would lead to prohibitive area and power overhead. Hence we choose to keep the RUI information in the memory instead of on chip. Therefore, in our case, the buffer size is bounded only by the memory size.

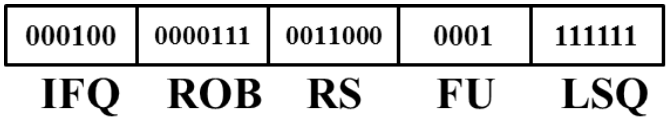


Figure 3.1: The RUI entry corresponding to an *add* instruction

3.2 Data Collection

We make use of the Reorder Buffer(ROB), a component in the superscalar architecuture, as a carrier for temporary RUI data. ROB is a hardware circular buffer implemented in the superscalar pipeline with one entry per dispatched yet not commited instruction [29]. We augment each ROB entry

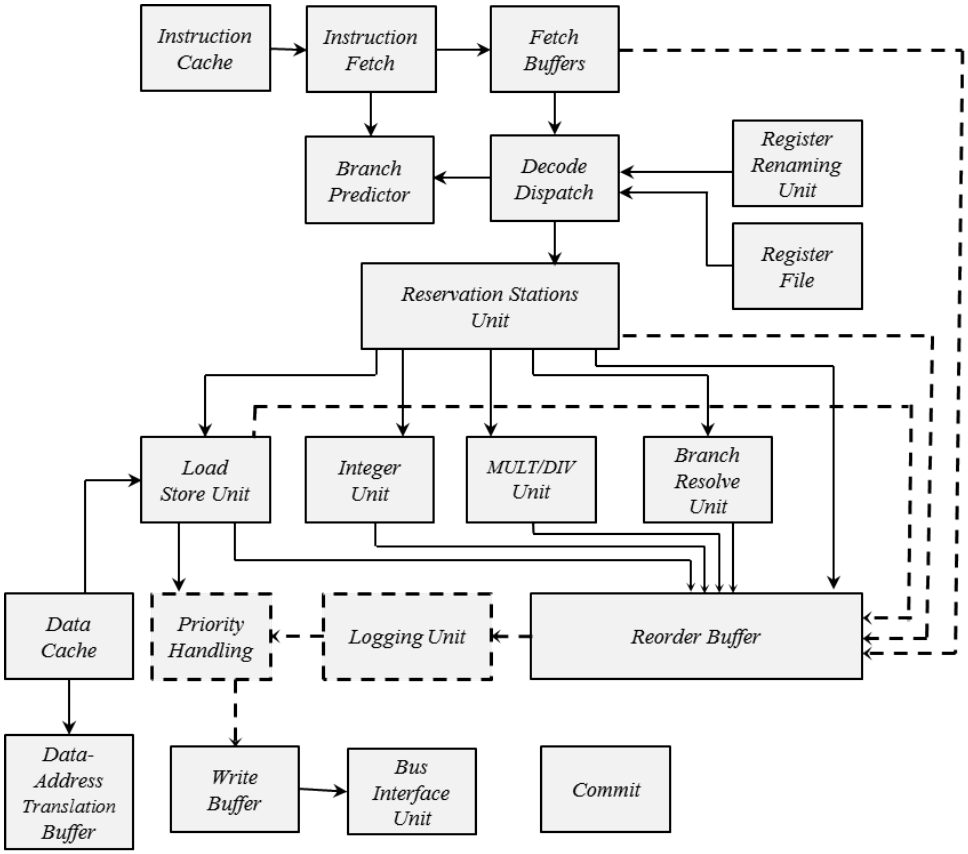


Figure 3.2: Organization of SCRIBE in the context of a processor derived from SuperScalar DLX processor [15]. Dashed lines show the added units and interconnects to implement SCRIBE.

with an X bit field ($X \propto lg(\textit{Total number of resources})$) to keep the RUI of the instruction corresponding to that entry. This field is filled with valid RUI as the instruction traverses the pipeline and makes use of specific resources. Since there is a one to one correspondence between each instruction and an ROB entry, the complete RUI of the instruction can be known when it reaches the commit stage. The RUIs are sent to the memory hierarchy when

their instructions are retired from ROB, and hence only correctly predicted instructions will be sent.

To gather information about which resources have been used by the corresponding instruction of that ROB entry, we need to connect the various stages of the pipeline with the ROB entry and send the information to the ROB at the appropriate time. The locations and the time for sending the information to the RUI field of an ROB entry is shown in table 3.1.

Resource	Where	When
IFQ	IFQ Head Pointer	Instruction Reading In Dispatch Stage
ROB	ROB Tail Pointer	ROB entry allocation In Dispatch Stage
LSQ	LSQ Tail Pointer	LSQ entry allocation in Dispatch Stage
RS	*Select* port of RS entry MUX	RS entry allocation In Dispatch Stage
Functional Unit	*Select* port of FU selection MUX	Issue Stage

Table 3.1: Details of the additional connections that must be introduced in the pipeline and when the information should be sent to the RUI

Figure 3.2 shows the whole processor with our mechanism added. The processor chosen to show the added components is a *Superscalar DLX* processor (based on [15]), with some modifications to make the figure more understandable. The units related to SCRIBE in the figure are the *logging* and *priority handling* units which are added to the commit stage of the pipeline. The logging unit is in charge of compressing the RUI entries and sending them to the priority handling unit. The priority handling unit in

each cycle chooses a regular store or a logging store to send to memory. These units are explained in detail in sections 3.3 and 3.4. The interconnects transferring the resource usage to the ROB are shown using dashed lines.

3.3 Logging Mechanism

As shown in figure 3.2, the ROB is connected to the logging unit which is in charge of sending the RUI to memory hierarchy for long time storage. The logging unit is expanded and shown in figure 3.3 along with the units communicating with it (*Priority handling*, LSQ and ROB). We explain the design of the logging mechanism in more detail below.

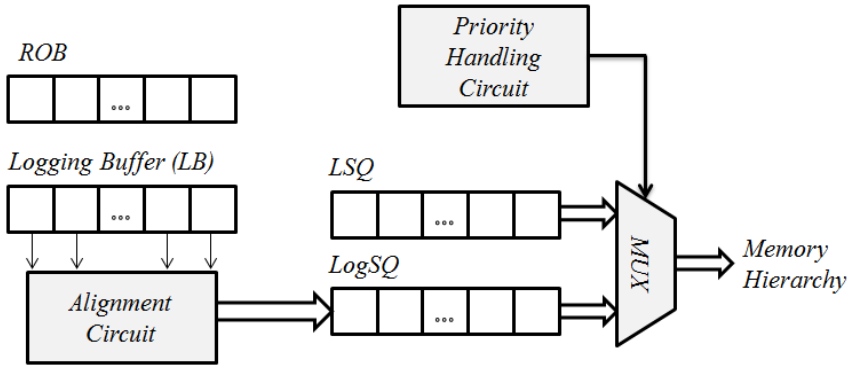


Figure 3.3: The *Logging Unit* includes the *Logging Buffer*, *Alignment Circuit* and *LogSQ*

When an instruction is retired from the ROB buffer, the RUI field of its ROB entry will be inserted into one of the partitions in the *Logging Buffer*. Logging Buffer (LB) is a dual partitioned queue and is in charge of keeping

the RUI of the retired instructions. Each of the partitions of the *LB* get filled separately. The role of the *alignment circuit* is to compress the RUI data and send them to the memory. When one of the partitions is full, its data is processed by *Alignment Circuit* and the other partition starts getting filled and vice versa. Thus, *data processing* and *filling* mode alternate with each other in each partition of the logging unit.

When a partition of the logging buffer gets full, the alignment circuits start concatenating RUI fields of multiple log buffer entries to form *quadwords* (64 bit words). These quadwords are then stored in *Logging Store Queue (LogSQ)* where they compete with the regular loads and stores of the program to be sent to memory hierarchy. This process is explained in section 3.4. If the LogSQ is full, the alignment circuits have to be stalled until a free entry in the logSQ becomes available. We discuss the implications of this stalling in Section 5.2.3.

3.4 Priority Handling

The goal of the priority handling unit is to mediate accesses to main memory between the stores performed by the logging mechanism and the regular stores performed by the processor. The priority handling unit includes the priority handling circuit and a multiplexer to select between the regular store instructions and the *logging store*s. The schematic of the simple priority handling circuit is depicted in figure 3.4. The priority handling unit takes inputs from the (regular) *load and store unit* and the *logging unit* and has its output connected to the *write buffer*.

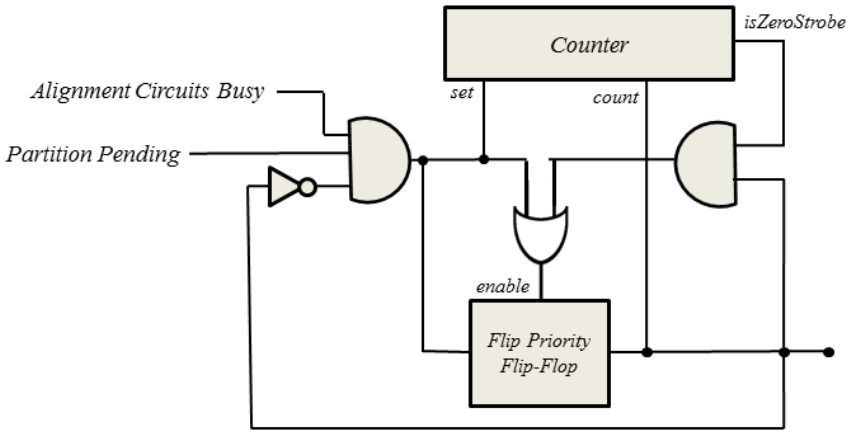


Figure 3.4: The priority handling circuit schematic

In the commit stage, when both a regular load/store instruction and a *logging store instruction* from *logSQ* are ready, one of them has to be chosen to be sent to memory hierarchy. If logging store instructions are not sent to memory on time, the logSQ becomes full and the alignment circuits are stalled. As described in section 3.3, the retiring process will be stalled in this situation which incurs a performance overhead for the processor. The main challenge is to reduce the probability of this situation occuring and hence reduce the performance overhead.

One way to avoid the logSQ from becoming full is to always prioritize the logging stores over regular ones. This will prevent the regular stores from blocking the draining process of logSQ. However, this will lead to stalling the regular non-store instructions which do not need the store port before a blocked regular store. This is because instructions are retired from the ROB in a FIFO (First-In-First-Out) manner. If logging store instructions

are always prioritized, a regular memory instruction at the head of the ROB would not be retired until the LogSQ is drained. This in turn will lead to stalling of the instruction commit mechanism in the processor, preventing the other partition of the logSQ from filling up with instructions. Our experiments show that this leads to severe performance slowdowns in the processor.

The other alternative is to prioritize regular stores over logging store instructions. However, this can lead to the processor being deadlocked due to the following sequence of events in the processor: ***(I)*** LogSQ becomes full and therefore one partition becomes full before processing of the other partition is finished. ***(II)*** As a result, the instruction retiring from the ROB will be stalled so that the logging mechanism can catch up. If there is a load/store effective address generation instruction at the head of ROB, it will also be held because the retiring process is stalled. ***(III)*** Since the held load/store has priority over logging store instructions, it will not let them be sent and so the logSQ will not drain. ***(IV)*** Hence, the alignment circuits will be kept stalled. This leads to a deadlock and the processor hangs.

Our solution is to use a hybrid approach where we switch the priorities between the logging stores and the regular stores based on the size of the LogSQ. We prioritize regular load/store instructions by default, until the logging mechanism starts stalling the commit stage (because of one partition becoming full before processing of the other one is finished). At this point, the logging store instructions gain priority over regular load/stores for approximately the number of cycles needed for logSQ to be drained. This value is computed at the time of the stall as: $\frac{|LogSQ|}{\#ofmemoryPorts}$. During all

other periods, the regular stores continue to get priority over the logging stores, and hence do not hold up the other instructions in pipeline.

3.5 Summary

In this chapter, we explain the details of the hardware part of our design, named SCRIBE. SCRIBE gathers the resources usage information (RUI) of each instruction in its corresponding Reorder Buffer entry. The length of an RUI field is around 50 bits but it could also slightly vary depending on the number of resources in the pipeline. RUI fields are sent to the *Logging Buffer* in SCRIBE when an instruction is retired from the pipeline. The RUI fields become aligned to form quad words and then are sent to a buffer called *Logging Store Queue*. The entries in the Logging Store Queue compete with the regular stores in the Load Store Queue for a memory interface. The *Priority Handling Circuit* in SCRIBE is responsible for defining which data should be sent to the memory in each cycle depending on the number of entries in each of the two buffers.

The next chapter explains how the software layer uses the RUI information provided by SCRIBE to find the faulty resource.

Chapter 4

SIED: Software Layer

In this section, we present SIED, the software portion of our technique.

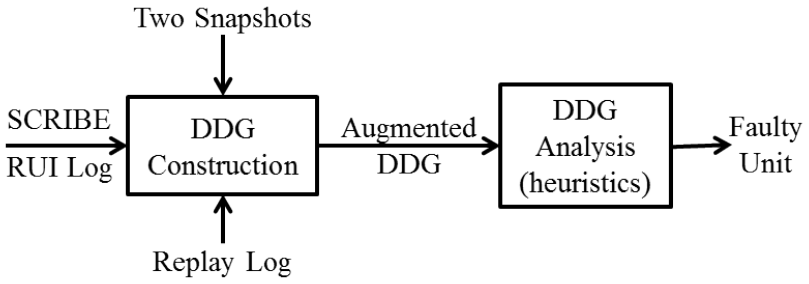


Figure 4.1: Flow of information during the diagnosis process

SIED is launched as a privileged process by the operating system on a separate core, which enables it to read the RUI segment in the failed program's memory written to by SCRIBE. Therefore, SIED has access to the history of dynamic instructions executed before the failure, and the micro-architectural resources used by those instructions.

Figure 4.1 shows the steps taken by SIED after a failure. First, the program is replayed on a separate core until the failed instruction, during which its DDG is built. The DDG is augmented with the RUI and the register/memory dumps from the original and replayed program executions.

This process is explained in Section 4.1. The augmented DDG is then fed to the DDG analysis step in Figure 4.1 which uses backtracking of DDG to find the candidates of the faulty resource. This process is explained in Section 4.2.

Example: We consider the program in Table 4.1 as a running example to explain the diagnosis steps. The example is drawn from execution of the benchmark *mcf* from SPEC 2006 benchmark suite on our simulator. However, some instructions have been removed from the real example to illustrate as many cases as possible in a compact way. As the program is executing, SCRIBE monitors the execution of instructions and logs their RUI to memory. The RUI logged by SCRIBE during the original execution is shown in Table 4.1 (the real RUI history includes a few thousands of entries; however, we only show the last few entries for brevity). For example, row #2 in Table 4.1 shows that the *store quadword* instruction has used entry 26 of the ROB, entry 15 of LSQ, entry 16 of IFQ, entry 11 of RS and functional unit 5 which is one of the memory ports (we consider memory ports as functional units).

Assume that in this example, the processor has multiple functional units, and the second functional unit (fu-1) is experiencing an intermittent fault that is triggered non-deterministically and lasts for several cycles. When the functional unit experiences the fault, one of the bits in its output becomes stuck at *zero* for this time period. This causes an incorrect value to be produced, as a result of which the program crashes. After the crash, the entire register and memory state of the process is dumped to memory. For this example, we only show the register and memory values produced by

#	Instruction	ROB	LSQ	IFQ	RS	FU
1	addi r1, -1, r1	25	-	15	16	1
2	stq r1, 400(r15)	26	15	16	11	5
3	bic r3, 16, r3	52	-	2	52	2
4	stl r3, 0(r9)	53	24	3	46	5
5	bis r31, r15, r30	84	-	1	7	1
6	ldq r1, 0(r30)	85	2	2	38	6
7	ldq r3, 8(r30)	86	3	3	19	6
8	ldq r30, 16(r30)	87	4	4	40	5
9	stq r5 , -32(r30)	88	5	5	44	6

Table 4.1: RUI of the instructions logged by SCRIBE. The original execution crashes at instruction 9.

the instructions in Table 4.1. These values are shown in Table 4.2, column "*Snapshot Original*". The "producer index" column represents the index of the instructions in Table 4.1 that last wrote to the locations in the second column.

Producer Index	Mem/Reg Location	Producer	Snapshot Original	Snapshot Replayed
2	0xd3e0	stq r1, 400(r15)	8	12
4	0xd988	stl r3, 0(r9)	10	10
6	r1	ldq r1, 0(r30)	16	0
7	r3	ldq r3, 8(r30)	8	20
8	r30	ldq r30, 16(r30)	20	0

Table 4.2: Snapshots: These represent the memory and register state dumps after the original and replayed executions

4.1 DDG Construction with RUI

As mentioned in Section 2.3, SIED uses deterministic replay techniques to replay the execution of the failed program and build its DDG. We refer to the first execution leading to the failure as the *original execution* and

the second execution performed by SIED as the *replayed execution*. The steps taken by SIED to build the DDG are as follows (step numbers below correspond to those in Figure 2.1):

1. The program is started from a previous checkpoint or from the beginning and replayed. However, the replayed program's control-flow may not match the control flow of the original execution, as the latter may have been modified by the intermittent fault. To facilitate fault diagnosis, the only difference between the original and the replayed execution should be the intermittent fault's effects on the registers and memory state. Therefore, the control flow of the replayed execution (target addresses of the *branch* instructions) is modified to match the original execution's control flow (step 4). To obtain the original execution's control flow, SCRIBE logs the branch target addresses of the program in addition to its RUI.

2. From the replayed execution, the information needed for building the Dynamic Dependence Graph (DDG) of the program is extracted and the DDG is built (step 5). Figure 4.2 shows the DDG for our example. The information required for building the DDG can be extracted by using a dynamic binary instrumentation tool (e.g. Pin [20]). We note that the overheads added by such tools would only be incurred during failure and subsequent diagnosis, and not during regular operation.

3. When the program flow of the replayed execution reaches the crash instruction (the instruction at which the original execution crashes), the register and memory state of the replayed execution is dumped to

memory (step 6). In the example, the replayed execution stops when reaching instruction 9 and the column "*snapshot replayed*" in Table 4.2 represents the register and memory state of the replayed program at that instruction.

4. The snapshots taken after the original and replayed executions are compared with each other to identify the final values that are different from each other. Because we assume a deterministic replay, any deviation in the values must be due to the fault. The producer instructions of these values are marked as *final erroneous* (or *final correct*) if the final values are different (or the same) in the DDG. The branch instructions that needed to be modified in step (i) to make the control flows match are also marked as *final erroneous* in the DDG. In the example, the values in the snapshot columns of Table 4.2 are compared, and the differences identified. The nodes corresponding to the instructions creating the mismatched values are marked in the DDG as final erroneous nodes (nodes 2, 6, 7 & 8), while node 4 with matching values, is marked as final correct.

5. The RUI of each instruction is added to its corresponding node in DDG. We call the resulting graph, the *augmented DDG*. The augmented DDG is used to find the faulty resource as shown in the next section.

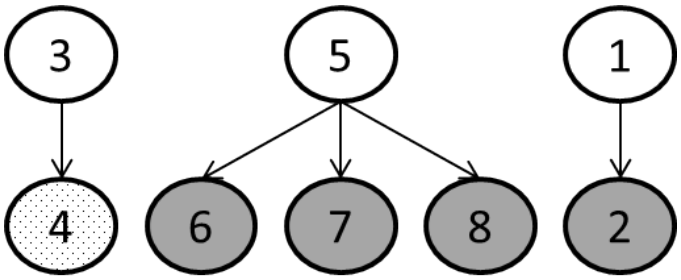


Figure 4.2: DDG of the program in the running example. Gray nodes are *final erroneous* and the dotted node is *final correct*

4.2 DDG Analysis

This section explains how SIED analyzes the augmented DDG to find the faulty resource. Because each dynamic instruction corresponds to a DDG node, we use the terms node and instruction interchangeably. The main idea is to start from final erroneous nodes in the augmented DDG (identified in Section 4.1), and backtrack to find nodes that have originated the error, i.e., the instructions that have used the faulty resource. The faulty resource is found by considering the intersection of the resources used by multiple instructions that have originated the errors. Recall that the list of resources used by an instruction is present in its corresponding node in the augmented DDG.

There are three types of nodes in the augmented DDG: **i)** Nodes that have used the faulty resource (originating nodes), **ii)** Nodes to which the error is propagated from an ancestor, **iii)** Nodes that have produced *correct* results (correct nodes). The goal of backtracking is to search for the originating nodes, by going backward from the final erroneous nodes (i.e.,

erroneous nodes in the final state), while avoiding the correct nodes. Naive backtracking does not avoid correct nodes, and because there can be many correct nodes in the backward slice of a final erroneous node, it will incur false-positives. Therefore, we propose two heuristics to narrow down the search space for the faulty resource based on the following observations:

1. If a final erroneous node has a correct ancestor node, the probability of the originating node being in the path connecting those two nodes is high. In other words, the faulty resource is more likely to be used in this path.

2. Having a final correct descendent decreases the probability that the node is erroneous.

3. Having an erroneous ancestor decreases the probability of the node being an originating node.

4. An erroneous node with all correct predecessors is an originating node.

Heuristics: To find faulty resources, each resource in the processor is assigned a counter which is initialized to *zero*. The counter of a resource is incremented if an instruction using that resource is likely to participate in creating an erroneous value, as determined by the heuristics. Resources having larger counter values are more likely to be faulty.

Algorithm 1 shows the pseudocode for heuristic 1. The main idea behind heuristic 1 is to examine the backward slices of the final erroneous nodes and increase the counter values of the appropriate resources based on the first three observations. In lines 3 to 8, for each final erroneous node n, the set

ALGORITHM 1: Heurisitc 1

```
input: resources
Algorithm heuristic1
1   foreach node n of final erroneous nodes do
2       Sn1 = Sn2 = φ // Initializing sets
3       foreach node k of n.ancestors do
4           if k.isLastCorrect() then
5               Sn1.add(getNodesBetween(n , k))
6       foreach R of resources do
7           if R is used in Sn1 then
8               counters[R]++;
9       foreach node k of nodes in backward slice of n do
10          if not(k.hasFinalCorrectDescendent) then
11              Sn2.add(k)
12      foreach R of resources do
13          if R is used in Sn2 then
14              if n.hasFaultyAncestor() then
15                  counters[R] += 0.5
16              else
17                  counters[R] += 1
```

S_{n1} is populated with the nodes between n and its final correct ancestors. The counters of the resources used by the nodes in the S_{n1} are incremented. Lines 9 to 11 correspond to the second observation. Every node in the backward slice of the final erroneous node n is added to set S_{n2} unless it has a final correct descendent. Finally, in lines 12 to 17, the nodes that are added to the set S_{n2} are checked to see if they have a faulty ancestor. If so, their counters are incremented by 0.5, and if not, the counters are incremented by 1. This is in line with the third observation that nodes with faulty ancestors are less likely to be originating nodes.

Algorithm 2 presents the second heuristic which is based on Observation 4. The algorithm starts from the final correct nodes and recursively marks the nodes that are likely to have produced correct output (lines 1 to 2).

ALGORITHM 2: Heurisitc 2

```
Procedure markCorrect(node n)
    foreach node p of the predecessors of n do
        ec ← p.getErroneousChildrenCount()
        if not (p.isErroneous() OR ec ≥ 2) then
            p.correct ← True
            markCorrect(p)
Procedure markErroneous(node n)
    foreach node p of the predecessors of n do
        cp ← p.getNonCorrectPredecessorsCount()
        cc ← p.getCorrectChildrenCount()
        if cp == 1 AND cc ≤ 1 then
            p.erroneous ← True
            markErroneous(p)
Algorithm heuristic2
1   foreach node n of the final correct nodes do
2       markCorrect(n)
3   foreach node n of the final erroneous nodes do
4       markErroneous(n)
5   foreach node n of the erroneous nodes do
6       cond1 ← (n.erroneousParentsCount == 0)
7       cond2 ← (n.correctParentsCount ≥ 1)
8       if cond1 AND cond2 then
9           Increment Counters of resources used in n;
```

Then it recursively marks the nodes that have likely produced erroneous outputs starting from the final erroneous nodes (lines 3 and 4). Finally, it checks all the erroneous nodes for the condition in the fourth observation i.e., being erroneous with no erroneous predecessor (lines 5 to 9). If the condition is satisfied, it increments the counters for the resources used by the erroneous nodes by 1.

After both heuristics are applied, the counter values computed by the heuristics are averaged to obtain the final counter values. The diagnosis algorithm identifies the top N_{deconf} resources with the highest counter values as candidates of the faulty resource, where N_{deconf} is a fixed value. These are the processor resources that are disabled to fix the intermittent fault after diagnosis. Thus N_{deconf} represents a trade-off between diagnosis accuracy and granularity. We study this trade-off in Section 5.2.1.

In general, we disable all the N_{deconf} resources identified by the diagnosis algorithm, with one exception. Because the number of functional units in a processor is typically low, we never disable more than one functional unit. This means that if the number of functional units among the resources with N_{deconf} highest final counter values is more than one, only the unit with the highest counter value is disabled.

Example: Due to space constraints, we only demonstrate the application of the first heuristic to the augmented DDG in Figure 4.2. Heuristic 1 starts from erroneous nodes (nodes 2, 6, 7 & 8). None of the erroneous nodes in this DDG have a final correct ancestor and therefore $S_{21} = S_{61} = S_{71} = S_{81} = \phi$. The backward slice for each of the erroneous nodes are collected by the algorithm ($S_{22} = \{2, 1\}$, $S_{82} = \{8, 5\}$,

$S_{72} = \{7, 5\}$, $S_{62} = \{6, 5\}$). The counters of resources in these sets are incremented by 1 as they have each participated in creating an erroneous value.

These nodes might *also* have participated in creating a final correct value. If so, they are pruned from the backward slice before their counters are incremented (Line 10). However, none of the nodes in the backward slices of the erroneous nodes in Figure 4.2 have final correct nodes as their children. Therefore, no *pruning* occurs in the example.

We can see that node 5 which has used the faulty resource *fu-1*, appears in the backward slices of three erroneous nodes (6, 7 & 8). This means that the counter related to *fu-1* is incremented 3 times. Meanwhile, *fu-1* is also used by the node 1 in the backward slice of erroneous node 2 (based on Table 4.1), and hence its counter value is again incremented by 1. The final counter values are shown in Table 4.3. As seen from the table, the faulty resource *fu-1* is the resource with the highest counter value of 4.

Resource	Value	Resource	Value
fu-1	4	fu-5	2
rob-84	3	rob-85	1
ifq-1	3	lsq-2	1
rs-7	3	...	1

Table 4.3: Counter values after applying heuristic 1 to DDG in Figure 4.2

Fault Recurrence: The above discussion considers a single occurrence of an intermittent fault. However, by their very definition, intermittent faults will recur, thus giving us an opportunity to diagnose them again. The above diagnosis process is repeated after every failure resulting from an intermittent fault, and each iteration of the process yields a different

counter value set. The final counter values are averaged across multiple iterations, thus boosting the diagnosis accuracy, and smoothing the effect of inaccuracies.

4.3 Summary

In this chapter, we explain the software part of the diagnosis technique named SIED. When a program crashes, SIED takes a snapshot of the program state, namely the registers and memory values. It then performs a deterministic replay of the program to the crash point on a non-faulty core. As the replay is being executed, the dynamic dependence graph of the program is extracted. When the replay finishes, another snapshot of the program is taken and compared with the snapshot after the original execution. The mismatched values are corrupted as a results of the intermittent fault. These values are marked as erroneous in the extracted DDG. The RUI information of each instruction is then added to its node in the DDG to form the *augmented DDG*. The augmented DDG is fed to the DDG analysis algorithms which are responsible for finding the faulty resource. These algorithms allocate a counter for each resource. Starting from the erroneous nodes, the algorithms backtrack the DDG and increment the counters of resources used in the instructions in backward slices of the erroneous nodes. The resources with the highest counter values at the end of this process are declared as the most likely ones to be faulty. In the next chapter, we describe our experimental methodology and the results of our evaluations.

Chapter 5

Evaluation

In this chapter, we present the experimental setup and the results of our evaluations. We answer the following research questions to evaluate our diagnosis technique:

1. **RQ 1**: What is the diagnosis accuracy or the probability that the technique correctly finds the faulty resource?

2. **RQ 2**: What is the performance overhead of repairing the processor after finding the faulty resource?

3. **RQ 3**: How much online performance, power and area overhead is incurred because of SCRIBE?

4. **RQ 4**: What is the offline performance overhead of SIED (Replay + DDG Construction and analysis)?

5. **RQ 5**: How does the accuracy and performance overhead increase in oracle mode (i.e. when the output of every instruction is stored along with its RUI)?

6. **RQ 6**: How does varying the length of buffers inside SCRIBE affect its performance overhead?

5.1 Methodology

5.1.1 SCRIBE

We implemented SCRIBE in *sim-mase*, a cycle-accurate micro-architectural simulator, which is a part of the SimpleScalar family of simulators [18]. We based our implementation on the SimpleScalar Alpha-Linux, developed as part of the XpScalar framework [9].

5.1.2 Configurations

To understand the overhead of our diagnosis mechanism across different processor families, we use three different configurations (Narrow, Medium and Wide pipelines) for our experiments. These respectively represent processors in the embedded, desktop and server domains, and similar configurations have been used in prior work on instruction-level duplication [30]. Table 5.1 lists the common configurations between the simulated processors and Table 5.2 shows the configurations that vary across processor families.

Parameter	Value
Level 1 Data Cache	32K, 4-way, LRU, 1-cycle latency
Level 1 Instruction Cache	32K, 4-way, LRU, 1-cycle latency
Level 2 combined data & instruction cache	512K, 4-way, LRU, 8-cycle latency
Branch Predictor	Bi-modal, 2-level
Instruction TLB	64K, 4-way, LRU
Data TLB	128K, 4-way, LRU
Memory Access Latency	200 CPU Cycles

Table 5.1: Common machine configurations

The RUI length is defined based on the type of the processor (recall

from Section 3 that *RUI Length* $\propto$ *lg*(*Total number of resources*)). We choose the LogSQ and Logging Buffer to be 32 and 64 entries respectively. Our experiments presented in section 5.2.6 indicate that increasing the sizes of these resources beyond 32 and 64 makes no significant improvement on performance.

Topic	Parameter	Machine Width		
		Narrow	*Medium*	*Wide*
Pipeline Width	Fetch	2	4	8
	Decode	2	4	8
	Issue	2	4	8
	Commit	2	4	8
Array Sizes	ROB Size	64	128	256
	LSQ Size	32	32	32
Number of of Functional Units	Integer Adder	2	4	8
	Integer Multiplier	1	1	1
	FP Adder	1	1	2
	FP Multiplier	1	1	1

Table 5.2: Different machine configurations

5.1.3 Benchmarks

We use eight benchmarks from the SPEC 2006 integer and floating-point benchmarks set. We chose these benchmarks as they were compatible with our infrastructure. We did not cherry-pick them based on the results.

5.1.4 Fault Injector

We extended *sim-mase* to build a detailed micro-architecture level fault injector. For each injection, the program is fast-forwarded 20 million instructions to remove initialization effects. Then a single intermittent fault burst

is injected into one of the following: **i)** Reorder Buffer entries, **ii)** Instruction Fetch Queue entries, **iii)** Reservation Station entries, **iv)** Load/Store Queue entries **v)** functional unit outputs. The starting cycle of the fault burst is uniformly distributed over the total number of cycles executed by the program. The number of cycles for which the fault persists (fault duration) is also uniformly distributed over the interval $[5, 2000]$, as voltage and temperature fluctuations last for around 5 to several thousands of cycles ([17, 28]).

After injecting the fault burst, the benchmark is executed and monitored for 1 million instructions to see if it crashes. We consider only faults that lead to crashes for diagnosis. This is because we do not assume the presence of error detectors in the program that can detect an error and halt it. To simulate a recurrent intermittent fault, we re-execute a benchmark up to 50 times while keeping the injection *location* unchanged. Note however that the starting cycle and fault duration are randomly chosen in each run. We report the results for scenarios in which 10 or more of the fault injections into a location led to crashes (out of 50 injections).

5.1.5 Diagnosis

SIED is implemented using Python scripts and starts whenever a benchmark crashes as a result of fault injection. We store the instruction outputs and extract the traces required to build the program's DDG by modifying the MASE simulator. However, these traces would be extracted by a virtual machine or a dynamic binary instrumentation tool in a real implementation of SIED (as explained in Section 4.1). The instruction outputs in a real

implementation can be stored by slightly modifying the presented SCRIBE structure. SIED also relies upon deterministic replay mechanisms (as explained in Section 2.3) for diagnosis. We have extended *sim-mase* to enable deterministic replay. Again, this would be implemented by a deterministic replay technique in a real implementation of SIED. We conducted the simulations and diagnosis experiments on an Intel Core i7 1.6GHz system with 8MB of cache.

5.1.6 Deconfiguration Overhead

The deconfiguration overhead is measured as the processor's slow-down after disabling the candidate locations of the faulty resource suggested by our diagnosis approach. We assume that the precise subset of resources suggested by our technique can be deconfigured. We used the medium width processor configuration from Table 5.2 for measuring the overhead after deconfiguration.

5.1.7 SCRIBE Performance, Power and Area Overhead

The performance overhead is measured as the percentage of extra cycles taken by the processor to run the benchmark programs when SCRIBE is enabled. For measuring the overhead, we execute each benchmark for 10^9 instructions in the MASE simulator [1]. We also implemented SCRIBE in the Wattch simulator [7] to evaluate its power overhead. The metric by which the power overhead of SCRIBE is evaluated is the average total power

[1] We do not use Simpoints due to incompatibilities between the benchmark format for the simulator and the format required by Simpoints.

per instruction. We used the CC3 power evaluation policy in Wattch as it also takes into account the fraction of power consumed when a unit is not used [7]. For on-chip area overhead, we implemented SCRIBE in the Illinois Verilog Model (an open-source Verilog implementation of a processor executing a subset of Alpha instruction set) [31]. The implementation of the SCRIBE circuits for estimating the on-chip area overhead is around 500 lines of Verilog code. The Illinois Verilog Model implementation alone is around 22000 lines of code.

The processor is then synthesized using Synopsys Design Compiler targeting TSMC 65nm technology and the area overhead of the components in SCRIBE is evaluated. We also perform timing analysis on the synthesized processor to test if adding SCRIBE increases the minimum clock cycle period required for the processor.

5.1.8 SCRIBE Oracle Mode

We also implemented SCRIBE in *Oracle mode*. In Oracle mode, SCRIBE stores the output of every instruction along with its RUI. Having the outputs of each instruction, the failed program is replayed and the outputs of the corresponding instructions in the two executions (*original* and *replayed*) get compared. The instructions with mismatching destination values are the ones affected by the fault. This set of instructions have either used the faulty resource (in which case we call them *faulty* instructions) or have inherited the fault from a faulty predecessor in DDG (In which case we call them *erroneous*). In other words, the difference between faulty and erroneous instructions is that erroneous instructions might not have used the

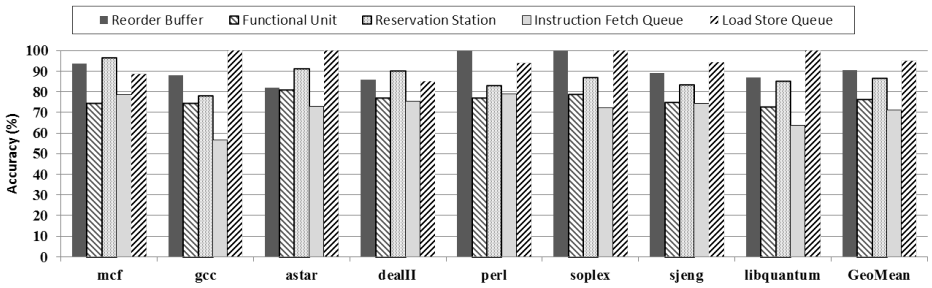


Figure 5.1: Accuracy results for applying the heuristics ($RN = 4$ and $N_{deconf} = 5$)

faulty resource but the error might have been propagated to them because of data flow between the instructions. The erroneous instructions which do not have any erroneous instructions as their parent in the DDG are *faulty* instructions.

5.2 Results

5.2.1 Diagnosis Accuracy (RQ 1)

Figure 5.1 shows the accuracy of our diagnosis approach for faults occurring in different units of the medium-width processor. We find that the average accuracy is 84% across all units. *To put this in perspective, our diagnosis approach identifies 5 resources out of more than 250 resources in the processor as faulty, and the actual faulty resource is among these 5 resources, 84% of the time (later, we explain why we chose 5).*

The diagnosis accuracy depends on the unit in which the fault occurs, and ranges from 71% for IFQ to 95% for LSQ. The reason for IFQ having

low accuracy is that faults in the IFQ cause the program to crash within a short interval of time (i.e., they have shorter crash distances). Short crash distances lead to lower accuracy, which is counter-intuitive as one expects longer crash distances to cause loss in the fault information and hence have lower accuracy. However, our DDG analysis algorithm explained in Section 4.2 uses backtracking the paths leading to final erroneous data. The more the number of these paths, the easier it is for our algorithm to distinguish the faulty resource from other resources, and hence higher the accuracy. Shorter crash distances mean fewer paths, and hence lower accuracy.

The main source of diagnosis inaccuracies is that SIED has only knowledge about *final* data (correctness of memory and register values at the *failure point*). The DDG analysis heuristics in Section 4.2 use backtracking from final erroneous data to speculate on the correctness of the data before the failure point. However, non-faulty resources are also used in the paths leading to final erroneous data, and can be incorrectly diagnosed as faulty by our technique.

One way to improve the diagnosis accuracy is to record the output of every instruction, thus eliminating the need for speculation on the correctness of the data before the failure point. However, our evaluations presented in section 5.2.5 shows that storing the output of every instruction imposes prohibitive performance overhead.

As explained in Section 4.2, SIED uses information from multiple occurrences of the intermittent fault to enhance the diagnosis accuracy. Let RN denote the number of recurrences of the failure, after which the diagnosis is performed. There is a trade-off among diagnosis accuracy and the failure

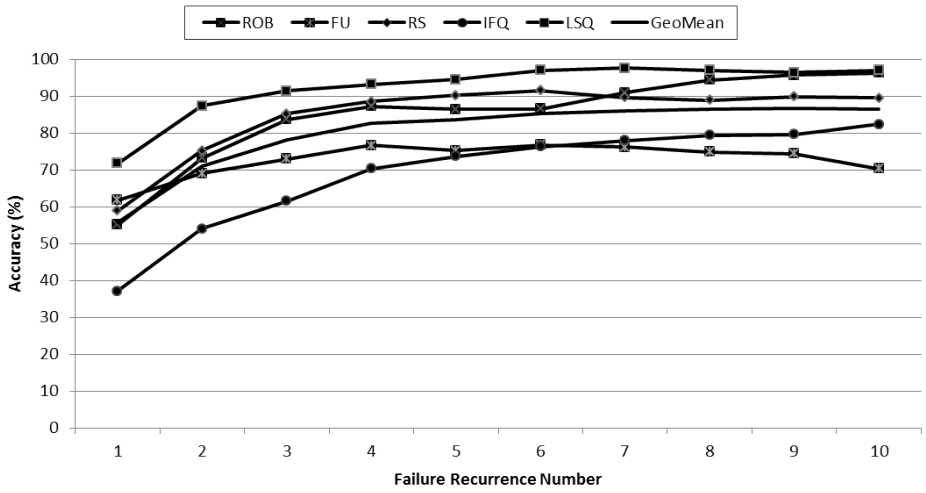


Figure 5.2: Average accuracy across benchmarks with respect to the number of failures ($N_{deconf} = 5$)

recurrence number (RN) for performing diagnosis. This means that diagnosis can be performed earlier at the expense of less accuracy or be postponed to receive more information from the subsequent failures and hence achieve higher accuracy, which in turn decreases the probability of the fault recurring after deconfiguration (and hence has lower overheads). Figure 5.2 shows how changing the RN value can affect the accuracy of diagnosis. We choose $RN = 4$ to perform diagnosis (Figure 5.1), as beyond this point, there is only a marginal increase in diagnosis accuracy with increase in RN.

5.2.2 Deconfiguration overhead (RQ 2)

As mentioned in section 4.2, N_{deconf} is the number of resources suggested by SIED as most likely to be faulty. *Diagnosis accuracy* is defined as the

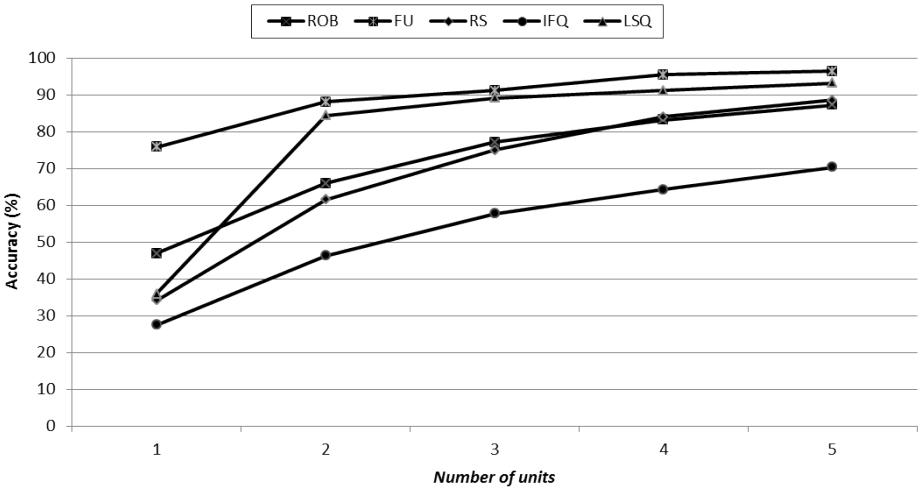


Figure 5.3: Accuracy with respect to N_{deconf} (RN = 4)

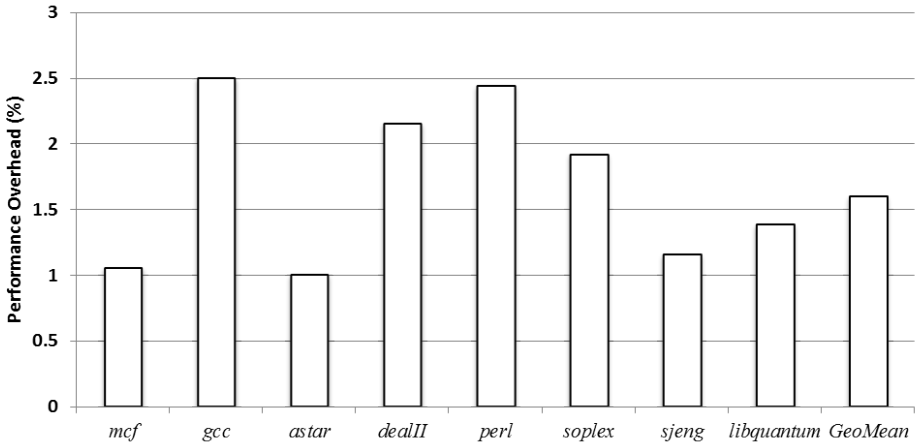


Figure 5.4: Performance overhead after deconfiguration ($N_{deconf} = 5$)

probability of the actual faulty resource being among the resources suggested by SIED. For the accuracies reported in Figure 5.1, N_{deconf} is chosen to be 5.

The processor is deconfigured after diagnosis by disabling these N_{deconf} resources. Although increasing N_{deconf} increases the likelihood of the processor being fixed after deconfiguration, it also makes the granularity of diagnosis more coarse-grained. In other words, by increasing N_{deconf}, deconfiguration disables more non-faulty resources along with the actual faulty resource. This results in performance loss after deconfiguration.

Figure 5.3 shows the accuracy of diagnosis as N_{deconf} varies from 1 to 5. As expected, increasing N_{deconf} increases the accuracy of diagnosis to 84% for $N_{deconf} = 5$. Figure 5.4 shows the average slowdown by disabling $N_{deconf} = 5$ resources suggested by our technique. As can be seen in the figure, the slowdown varies from 1% to 2.5%, with an average of 1.6%. This shows that disabling $N_{deconf} = 5$ resources only incurs a modest performance overhead after reconfiguration, and hence we choose this value.

5.2.3 SCRIBE Performance, Power and Area Overhead (RQ 3)

Figure 5.6 shows the performance overhead incurred by SCRIBE across three processor configurations, narrow, medium and wide, described in Section 5.1. The *geometric mean* of the overheads across all configurations is 14.7%. In all but one case (except *soplex*), the *wide* configuration ($GeoMean$ = 23.21%) incurs higher overhead than the *medium* ($GeoMean$ = 11.88%) and *narrow* ($GeoMean$ = 11.53%) configurations. The *Medium* and *narrow* configurations are comparable in terms of overhead. The *wide* processor has high overhead as it is able to utilize the resources better, thus leaving fewer free slots to be used by SCRIBE for sending logging stores to memory.

Figure 5.5 shows the correlation between the performance overhead for each benchmark and the baseline IPC of the benchmark. There is a positive correlation between the IPC and the overhead (i.e. overhead increases as the baseline IPC increases). For example, the baseline IPC values for *perl* and *libquantum* are 1.35 and 3 and their overheads are 6.18% and 28.05% respectively. In other words, SCRIBE uses the opportunities made by the resources being underutilized to perform its logging.

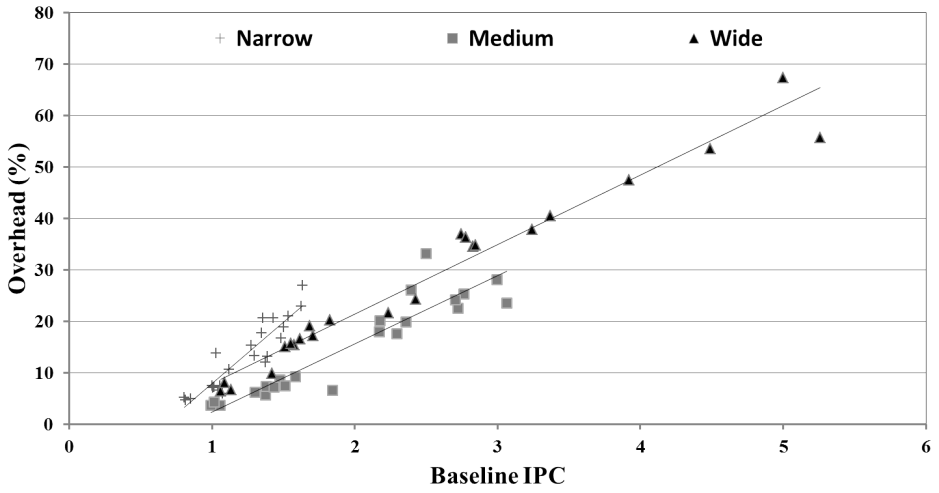


Figure 5.5: Correlation between baseline IPC and performance overhead. The trendline for each dataset is also shown.

To better understand the performance overheads incurred by SCRIBE, we break down the overhead into four parts. Figure 5.7 shows the overhead breakdown of SCRIBE for the medium width processor. The overhead components are:

- **Memory Ports Pressure**: Since multiple *Logging Store*s are being

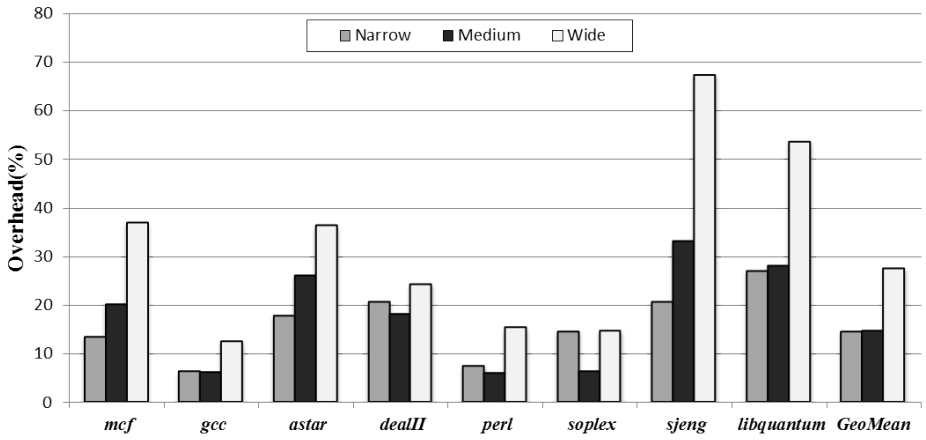


Figure 5.6: The performance overhead of SCRIBE applied to three configurations: *Narrow*, *Medium* and *Wide*

sent to the memory hierarchy by SCRIBE, the memory ports might get busy in some cycles when needed by regular loads/stores. This would make the regular loads and stores stall. This component is responsible for $\sim 41\%$ of the total SCRIBE overhead.

- **Stalling Regular Stores**: In the cycles in which the logging mechanism has priority, regular stores at the head of ROB will be stalled and hence the commit stage becomes stalled. This will continue until the regular stores gain priority over *logging store*s again. This component is responsible for $\sim 32\%$ of the total SCRIBE overhead.

- **Reducing Commit Bandwidth**: The commit stage has a limited bandwidth. SCRIBE consumes part of this bandwidth in each cycle by attempting to send logging stores to the memory hierarchy. This

part of the overhead is responsible for $\sim 23\%$ of the overhead.

- **Stalling the Commit Stage**: This happens when one partition of the *Log Buffer* is not still done being compressed while the other partition becomes full. In this case, the commit stage will be stalled. This component is responsible for only $\sim 4\%$ of the total SCRIBE overhead.

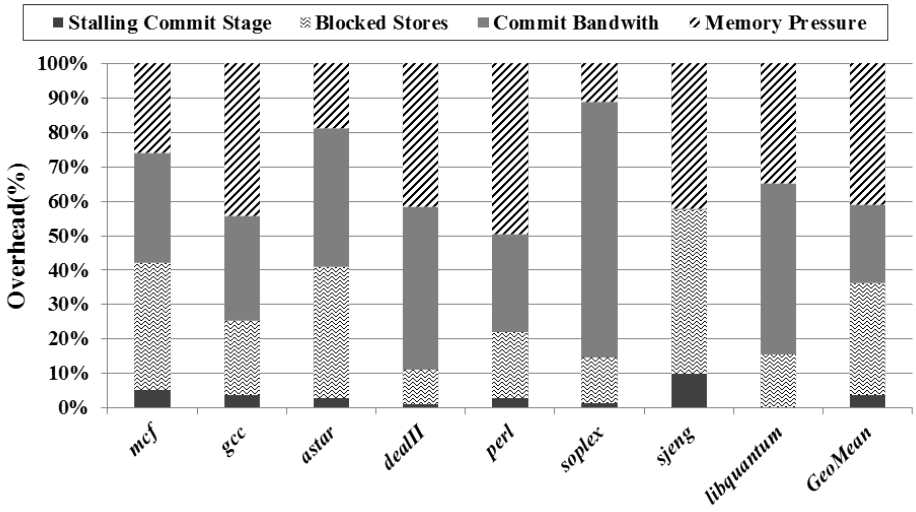


Figure 5.7: The breakdown of the performance overhead for medium width processor

Thus we see that the overhead of SCRIBE is dominated by the memory pressure it introduces to the processor, as well as the stalling of regular stores in the processor.

The results of the timing analysis after synthesizing the design shows that the SCRIBE is not in the critical path of the processor and therefore does not increase the minimum clock cycle period of the processor.

As far as power is concerned, SCRIBE has 9.3% power overhead on average. This includes both active power and idle power. Figure 5.8 shows the breakdown of the power consumption overhead. As seen in the figure, only 7.9% of the extra power is used by the components of SCRIBE. The rest of the power overhead is due to the extra accesses to the D-Cache and the extra cycles due to SCRIBE (indicated in the figure as *Other Components*).

The synthesis reports from the Synopsys Design Compiler show that SCRIBE components have only 0.95% of on-chip area overhead. The small on-chip area overhead is expected as SCRIBE only adds around 1.75 Kbytes of storage to the processor. A study synthesizing a comparable technique has reported adding 50Kbytes of on-chip storage and having 2% area overhead [23]. Figure 5.9 shows the break down of the area overhead of SCRIBE components. As seen from the figure, the area is largely dominated by the hardware array structures. The logging buffer with 64 entries has the highest area overhead (65.1%). This is because the logging buffer has the largest number of entries between the buffers used in SCRIBE. The priority handling circuit on the other hand has the least area overhead (0.1%) as it is simple and consists of very few components (Figure 3.4).

5.2.4 SIED Offline Performance Overhead (RQ 4)

This overhead consists of: **i)** Replay time **ii)** DDG construction and analysis time. The average replay time depends on the program and whether it is replayed from a checkpoint or from the beginning. We do not consider this time as it depends on the checkpointing interval. The DDG construction and analysis time took 2 seconds on average, for our benchmarks.

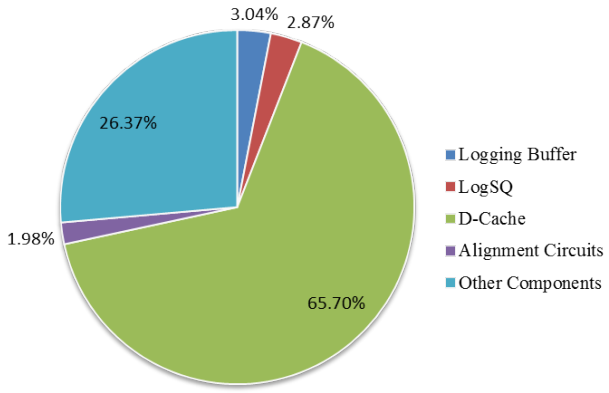


Figure 5.8: The breakdown of power consumption of SCRIBE

5.2.5 SCRIBE Oracle Mode (RQ 5)

As mentioned in 5.1.8, we also implemented SCRIBE in *Oracle mode* in which the output of every instruction is sent to memory along with its RUI. Figure 5.10 shows the accuracy of the oracle mode for four of the SPEC 2006 benchmarks. The average accuracy across all the units in the Oracle mode is 97.8%. with the minimum accuracy belonging to the Instruction Fetch Queue (92.3%).

The relative inaccuracy of IFQ stems from the fact that some of the faults hitting IFQ entries invalidate the instruction leading to immediate crashes in the decoded stage and before being dispatched to the ROB. Therefore the faulty instruction is not visible in the DDG and can not be used to find the faulty resources. However, the mentioned hardware structure and our implementation does not catch these instructions. Another source of inaccuracy for the oracle is long crash distances. This means that the faulty

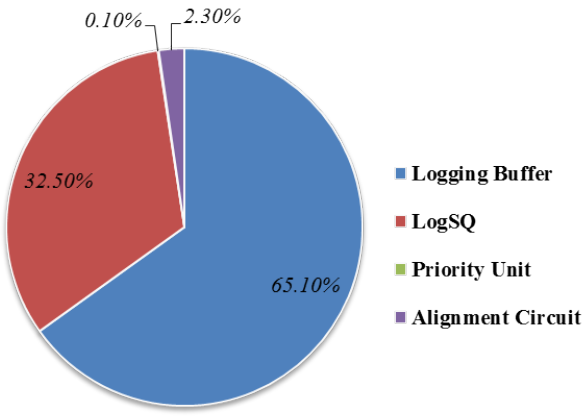


Figure 5.9: Area overhead breakdown of SCRIBE

resource (the fault origin) is outside of the range of the recorded instructions. Therefore, backtracking erroneous instructions only reaches us to the oldest erroneous instruction available which is not necessarily a faulty instruction. Increasing the size of the RUI Buffer for the Oracle mode eliminates this source of inaccuracy.

We can see from figure 5.10 that the the Oracle mode is very accurate. The main reason for the high accuracy of the oracle mode in comparison with our proposed technique is that with the oracle mode, the DDG analysis module has the knowledge of correctness of all the data (nodes) in the backward slices of the DDG. Having this information, backtracking can be continued until there is no erroneous predecessor for an erroneous node. The node without an immediate erroneous predecessor has used the faulty resource. This means that the DDG analysis is aware of the exact instructions which used the faulty resource. This important information is absent during

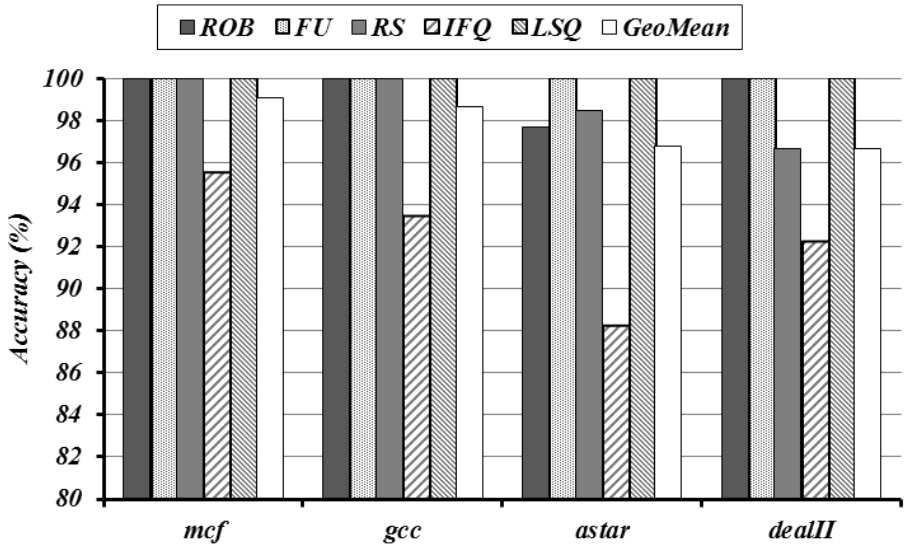


Figure 5.10: Accuracy results for the *Oracle mode* of SCRIBE ($RN = 4$ and $N_{deconf} = 5$)

the DDG analysis without the oracle mode. The DDG analysis without the oracle mode only has access to the correctness of the data in the crash state (which it aquires by comparing the snapshots taken after the *original* and the *replayed* executions). Therefore, it considers all of the nodes in the backward slices of the erroneous nodes as the instructions which might have used the faulty resource and therefore increases their counters. This assumption is not always correct and leads to some inaccuracy in the diagnosis.

In order to reach to the high accuracy of the oracle mode, we need to have the output data of all the instructions to achieve this high accuracy. Figure 5.11 shows the performance overhead of storing the output data of every instruction, for 32-bit instructions and 64-bit instructions, for three SPEC 2006 programs. The overhead for storing 0 extra bits corresponds to that

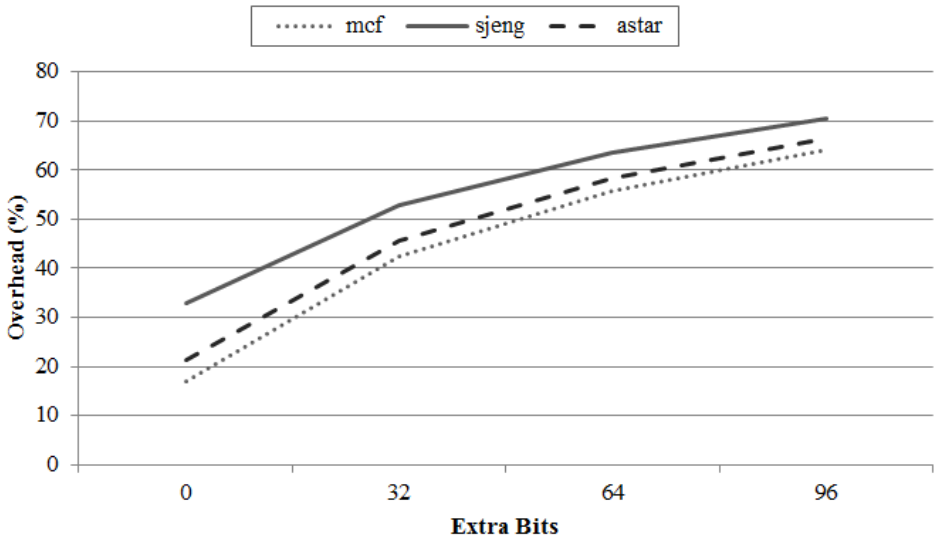


Figure 5.11: Performance overhead of sending the destination register values of every instruction on performance overhead (0 bits corresponds to only sending the RUI as in our technique)

of storing only the resource usage bits, as done by our technique (explained in Section 3). As seen from the Figure 5.11, the overheads for storing the results of 32 and 64 bit instructions are respectively 2X and 3X that of the overhead of only storing the resource usage information. Therefore, we choose not to record the output of every instruction for diagnosis.

5.2.6 Sensitivity Study of Buffer Sizes (RQ 6)

We perform a sensitivity analysis of the performance overhead of SCRIBE by varying the sizes of Logging Buffer and LogSQ. We use the medium width processor from table 5.2 in this part of the evaluation. For conciseness we use only three of the nine benchmarks, though they are representative of the

general trend we observed.

Figures 5.12 and 5.13 show the variation of the overhead with the sizes of the Logging Buffer and LogSQ. As can be seen in the figures, the overhead initially decreases as we increase the sizes of the buffers. However, increasing the sizes of Logging Buffer and LogSQ beyond 64 and 32 entries respectively does not provide any benefit. This is why we choose values of 64 and 32 for the sizes of the Logging Buffer and the LogSQ respectively.

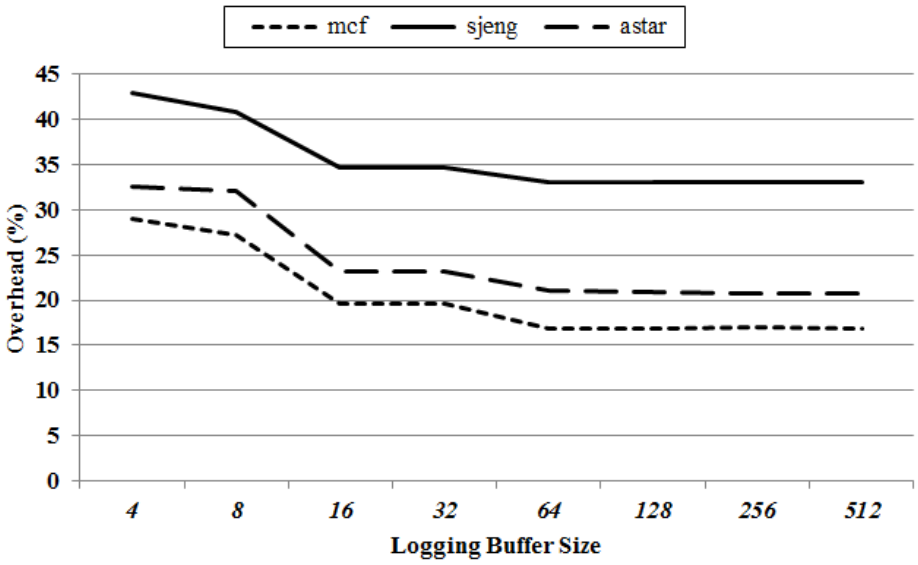


Figure 5.12: The overhead with respect to logging buffer size for medium width processor

5.3 Summary

In this chapter, we described our experimental setup and the results of our evaluations. We evaluate the accuracy of our technique as well as different

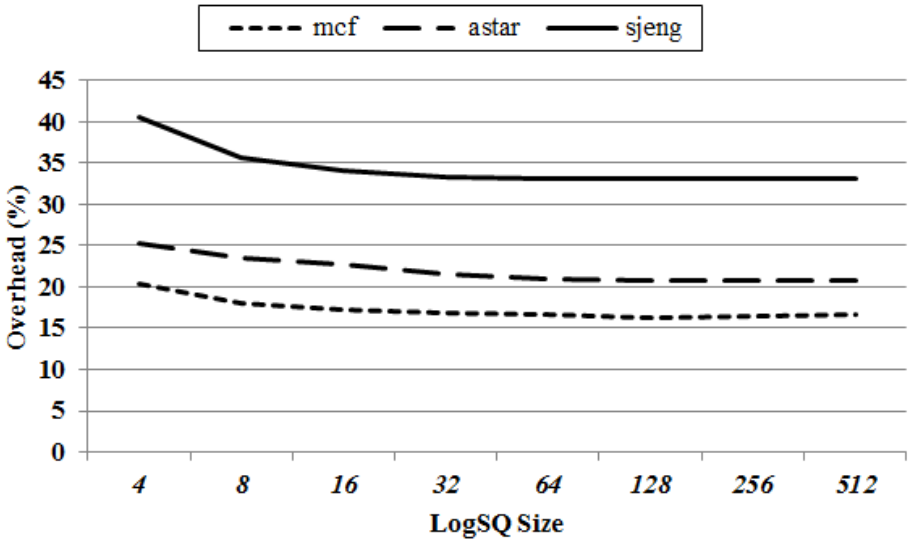


Figure 5.13: The overhead with respect to logSQ size for medium width processor

overheads of our design. Our synthesis results show that SCRIBE has 0.95% on-chip area overhead. The results of our simulation experiments show that SCRIBE has a performance overhead of 12% and power overhead of 9%, on average. Our fault injection experiments show that our technique has an average accuracy of 84% in diagnosing the faulty resource, which in turn enables fine-grained deconfiguration with less than 2% performance loss after deconfiguration.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

Hardware intermittent faults are becoming more prominent as a side effect of technology scaling. The diagnosis of faults is essential for automatic repair of the chip using deconfiguration. The granularity of diagnosis is related with the amount of performance loss after repair. In other words, the finer the granularity of diagnosis, the less performance will be lost after deconfiguration. Furthermore, since the intermittent faults can also be caused by temperature variations and the wear-out due to aging, an intermittent fault diagnosis mechanism needs to be present during the operational settings of a chip. This in turn, requires such a mechanism to incur as low amount of overhead as possible.

In this thesis, we proposed a hardware/software integrated scheme for diagnosing intermittent faults in processors. Our scheme consists of SCRIBE, the hardware layer which enables fine-grained software layer diagnosis and SIED, the software layer which uses the information provided by SCRIBE after a failure to diagnose the intermittent fault. The information provided by SCRIBE consists of the detailed micro-architectural resource usage of each instruction in the pipeline. We found that using SCRIBE and SIED,

the faulty resource can be correctly diagnosed in 84% of the cases on average. Our scheme incurs about 12% performance overhead, and about 9% power consumption overhead and less than 0.95% on-chip area overhead (for a desktop class processor). The fine granularity of our diagnosis mechanism enables automatic repair of the chip using deconfiguration, with less than 2% slow-down on average.

To the best of our knowledge, this is the first study to decouple the mechanisms for gathering the information required by diagnosis and the module performing the diagnosis operations. The hardware part of the technique can be used as a building block for the other diagnosis mechanisms. In other words, the resource usage information provided by the hardware layer could be used by the other software layer diagnosis algorithms.

6.2 Future Work

The work presented in this thesis can be extended from several aspects. The hardware part of the design, could be continued to explore micro-architectural techniques to further optimize the overhead of SCRIBE. This is the first study to separate the mechanisms for diagnosis information gathering from the diagnosis algorithms. Hence, other software layer diagnosis techniques relying on the resource usage information of instructions could also be designed to enhance the accuracy of diagnosis.

Intermittent faults are not always present during the whole period of the operation of a chip. A technique for only turning the SCRIBE on when intermittent faults are present could significantly decrease the overall online

performance and power overhead of SCRIBE.

The hardware layer presented in this thesis is able to provide the detailed information about the pipeline resources used by the instructions. Other applications such as hardware performance monitoring tools could also benefit from this information provided by SCRIBE.

The measurement of the overhead in this work is based on simulation. The evaluation of the technique could be extended by running workloads on a real synthesized chip equipped with SCRIBE. The accuracy measurement could also be extended to consider other fault models such as multiple fault locations over time.

Bibliography

[1] Hiralal Agrawal and Joseph R. Horgan. Dynamic program slicing. PLDI, pages 246–256, 1990.

[2] T.M. Austin. DIVA: A reliable substrate for deep submicron microarchitecture design. MICRO, pages 196–207, 1999.

[3] A. Bondavalli, S. Chiaradonna, F. di Giandomenico, and F. Grandoni. Threshold-based mechanisms to discriminate transient from intermittent faults. *IEEE Transactions on Computers*, 49:230–245, Mar 2000.

[4] Shekhar Borkar. Microarchitecture and design challenges for gigascale integration. In *Keynote Speech, 37th International Symposium on Microarchitecture*, MICRO, 2004.

[5] Shekhar Borkar, Tanay Karnik, Siva Narendra, Jim Tschanz, Ali Keshavarzi, and Vivek De. Parameter variations and impact on circuits and microarchitecture. DAC, pages 338–342, 2003.

[6] Fred A. Bower, Daniel J. Sorin, and Sule Ozev. A mechanism for online diagnosis of hard faults in microprocessors. MICRO, pages 197–208, 2005.

[7] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. ISCA, pages 83–94, 2000.

[8] J. Carretero, X. Vera, J. Abella, T. Ramirez, M. Monchiero, and A. Gonzalez. Hardware/software-based diagnosis of load-store queues using expandable activity logs. HPCA, pages 321–331, 2011.

[9] N.K. Choudhary, S.V. Wadhavkar, T.A. Shah, H. Mayukh, J. Gandhi, B.H. Dwiel, S. Navada, H.H. Najaf-abadi, and E. Rotenberg. FabScalar: Composing synthesizable RTL designs of arbitrary cores within a canonical superscalar template. ISCA, pages 11–22, 2011.

[10] C. Constantinescu. Intermittent faults and effects on reliability of integrated circuits. RAMS, pages 370–374, 2008.

[11] Cristian Constantinescu. Trends and challenges in VLSI circuit reliability. *IEEE Micro*, 23(4):14–19, 2003.

[12] Kypros Constantinides, Onur Mutlu, Todd Austin, and Valeria Bertacco. Software-based online detection of hardware defects: Mechanisms, architectural support, and evaluation. MICRO, pages 97–108, 2007.

[13] Andrew DeOrio, Qingkun Li, Matthew Burgess, and Valeria Bertacco. Machine learning-based anomaly detection for post-silicon bug diagnosis. DATE, pages 491–496, 2013.

[14] George W. Dunlap, Dominic G. Lucchetti, Michael A. Fetterman, and

Peter M. Chen. Execution replay of multiprocessor virtual machines. VEE, pages 121–130, 2008.

[15] H. Eveking. Superscalar dlx documentation. `http://www.rs.tu-darmstadt.de/downloads/docu/dlxdocu/DlxPdf.zip`.

[16] S. Gupta, Shuguang Feng, A. Ansari, and S. Mahlke. StageNet: A reconfigurable fabric for constructing dependable CMPs. *IEEE Transactions on Computers*, 60(1):5–19, Jan 2011.

[17] R. Joseph, D. Brooks, and M. Martonosi. Control techniques to eliminate voltage emergencies in high performance processors. HPCA, pages 79–90, 2003.

[18] E. Larson, S. Chatterjee, and T. Austin. MASE: a novel infrastructure for detailed microarchitectural modeling. ISPASS, pages 1–9, 2001.

[19] Man-Lap Li, P. Ramachandran, S.K. Sahoo, S.V. Adve, V.S. Adve, and Yuanyuan Zhou. Trace-based microarchitecture-level diagnosis of permanent hardware faults. DSN, pages 22–31, 2008.

[20] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. PLDI, pages 190–200, 2005.

[21] J. W. McPherson. Reliability challenges for 45nm and beyond. DAC, pages 176–181, 2006.

[22] Edmund B. Nightingale, John R. Douceur, and Vince Orgovan. Cycles, cells and platters: An empirical analysis of hardware failures on a million consumer PCs. EuroSys, pages 343–356, 2011.

[23] Sung-Boem Park and S Mitra. IFRA: Instruction footprint recording and analysis for post-silicon bug localization in processors. DAC, pages 373–378, 2008.

[24] Andrea Pellegrini and Valeria Bertacco. Application-aware diagnosis of runtime hardware faults. ICCAD, pages 487–492, 2010.

[25] L. Rashid, K. Pattabiraman, and S. Gopalakrishnan. Intermittent hardware errors recovery: Modeling and evaluation. QEST, pages 220–229, 2012.

[26] Layali Rashid et al. Modeling the propagation of intermittent hardware faults in programs. In *Dependable Computing (PRDC), 2010 IEEE 16th Pacific Rim International Symposium on*, pages 19–26. IEEE, 2010.

[27] Giacinto P. Saggese, Nicholas J. Wang, Zbigniew T. Kalbarczyk, Sanjay J. Patel, and Ravishankar K. Iyer. An experimental study of soft errors in microprocessors. *IEEE Micro*, 25(6):30–39, 2005.

[28] Kevin Skadron, Mircea R. Stan, Karthik Sankaranarayanan, Wei Huang, Sivakumar Velusamy, and David Tarjan. Temperature-aware microarchitecture: Modeling and implementation. *ACM Trans. Archit. Code Optim.*, 1(1):94–125, Mar 2004.

[29] J.E. Smith and G.S. Sohi. The microarchitecture of superscalar processors. *Proceedings of the IEEE*, 1995.

[30] A. Timor, A. Mendelson, Y. Birk, and N. Suri. Using underutilized CPU resources to enhance its reliability. *IEEE Transactions on Dependable and Secure Computing*, 7(1):94–109, Jan 2010.

[31] N.J. Wang, J. Quek, T.M. Rafacz, and S.J. Patel. Characterizing the effects of transient faults on a high-performance processor pipeline. DSN, pages 61–70, June 2004.

[32] C. Weaver and T. Austin. A fault tolerant approach to microprocessor design. DSN, pages 411–420, 2001.

[33] Philip M. Wells, Koushik Chakraborty, and Gurindar S. Sohi. Adapting to intermittent faults in multicore systems. ASPLOS, pages 255–264, 2008.

[34] M. Xu, R. Bodik, and M.D. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. ISCA, pages 122–133, 2003.