ABSTRACT

REDDY, VIMAL KODANDARAMA. Exploiting Microarchitecture Insights for Efficient Fault Tolerance. (Under the direction of Dr. Eric Rotenberg).

Technology scaling makes transistors more susceptible to transient faults. As a result, it is becoming increasingly important to incorporate transient fault tolerance in future processors. Traditional transient fault tolerance approaches duplicate in time or space for robust fault tolerance, but are expensive in terms of performance, area, and power, counteracting the very benefits of technology scaling. To make fault tolerance viable for commodity processors, unconventional techniques are needed that provide significant fault protection in an efficient manner. In this spirit, this thesis presents two low-overhead approaches to fault tolerance based on microarchitecture insights.

First, prediction-based partial redundant threading (PRT) is presented as a low-overhead alternative to full redundant multithreading (RMT). In RMT, two copies of a program are executed on a simultaneous multithreading (SMT) substrate. Outcomes of duplicated instructions are compared to detect transient faults in the processor. RMT incurs high performance and power overheads due to full redundant execution (as high as 40% slowdown). In prediction-based PRT, confident predictions are leveraged as effective proxies for redundant execution, based on the idea that a correct prediction of an instruction's outcome is the same as the outcome produced by fault-free execution of the instruction. Confidently-predicted instructions and their producers are skipped in the redundant thread (as many as 57% instructions skipped). This predictive thread is shown to be as effective as a full thread for checking purposes, but much more efficient.

Second, a superscalar processor is designed with built-in checks that indirectly detect low-level transient faults, by observing microarchitecture-level anomalies they cause. A single check covers many logic blocks, similar in spirit to outcome checks in RMT, but without the overheads of redundant execution. This dissertation develops several novel microarchitecture-level fault checks for protecting critical superscalar processor structures. Most notably, 1) inherent time redundancy (ITR) exploits program repetition to detect faults in decode signals, thereby covering the fetch and decode units, 2) register name authentication (RNA) asserts consistencies among renaming structures to detect faults affecting register renaming, and 3) timestamp-based assertion checking (TAC) asserts sequential order among dependent instructions to detect faults affecting dynamic instruction scheduling. Based on these checks, a fault-checking regimen is engaged to comprehensively protect a superscalar processor pipeline. To evaluate fault tolerance of the processor, a new fault injection strategy is developed. It involves analyzing the microarchitecture of a superscalar processor in depth and identifying high-level faults which can be modeled in a timing simulator, enabling a fast and reasonably accurate evaluation. Exclusive fault injection experiments reveal that the new fault-checking regimen provides substantial fault coverage to the processor, making the case for a canonical fault-tolerant superscalar processor.

Exploiting Microarchitecture Insights for Efficient Fault Tolerance

by

VIMAL KODANDARAMA REDDY

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Engineering

Raleigh, North Carolina

2007

APPROVED BY:

____

Dr. Eric Rotenberg, Chair of Advisory Committee

__________________________	__________________________
Dr. Thomas M. Conte	Dr. Suleyman Sair

____

Dr. Warren Jasper

DEDICATION

I dedicate this work to my wife, Lakshmi,

and my parents, Prof. K. K. Reddy and Mrs. Vijiram.

BIOGRAPHY

Vimal Kodandarama Reddy was born on November 3, 1977, in the garden city of India, Bangalore, to Prof. Kodandarama Reddy and Mrs. Vijiram. He went to high school at Army High School, Bangalore, and completed his pre-university education at St. Joseph's Arts and Science College, Bangalore. He earned his Bachelor's degree in Electronics and Communications from B.M. Sreenivasiah College of Engineering, Bangalore, in 1999. He worked as a software engineer at Siemens Communications Software, Bangalore, from 1999 to 2001. In fall of 2001, he began to pursue graduate studies at North Carolina State University, Raleigh, North Carolina. He earned his Master's degree in Computer Engineering with a thesis in the summer of 2003, and after that, began work on his doctoral degree under the guidance of Prof. Eric Rotenberg, working in the areas of high-performance architectures and fault-tolerant architectures. During his graduate studies, he had the opportunity to intern at IBM, Raleigh, during the summer of 2002, and Intel Corporation, Hudson, during the spring and summer of 2005.

ACKNOWLEDGMENTS

The pursuit of my doctoral degree has been a fun and rewarding journey. Here, I want to thank everyone who helped me reach my destination.

First and foremost, I want to thank my parents, Prof. Kodandarama Reddy and Mrs. Vijiram. I could not have succeeded without their unconditional love, support, and prayers.

I am indebted to my advisor Prof. Eric Rotenberg for inspiring me to do computer architecture research and training me to become a good researcher. Eric has been a great mentor and friend. I thank him for giving me the freedom to work on many projects, while always making sure I was on the right track. His high standards, work ethics, attention to quality and emphasis on novelty and individuality, and most importantly, his sense of humor, have influenced my thinking and personality to a great degree. I feel privileged and proud to have been advised by him.

I wish to thank my committee members Prof. Tom Conte, Prof. Suleyman Sair, and Prof. Warren Jasper. Their guidance and expertise improved the quality of this dissertation.

I would like to express my deepest gratitude to my wife Lakshmi for her love, compassion and support. Her companionship and cheerfulness were my solace during many uncertain periods that graduate school brought with it. I really thank her for the many selfless sacrifices she made for my cause, and the patience and trust she has shown in me.

I thank my brother Vikram Reddy for his encouragment and wish him good luck in his endeavors. I thank my cousin Hari, and his wife Neema, for always being there for me and helping me out in various ways.

I feel privileged to have crossed paths with several wonderful people in the CESR lab. Thanks to Ahmed Al-Zawawi for being a great friend, mentor and an excellent sounding board to my ideas, regardless of their merit. I benefited immensely from Ahmed's wisdom and experience. Many problems related to my research were resolved after brainstorming with him. I learnt several valuable lessons from him: asking good questions, how to not get ripped off, bargaining, winning an argument (by actually losing many to him), and finally, finding great deals, that saved me lots of dough :) I really wish him good luck in his endeavors. Thanks to Balaji, Chad, Hashem, Mark, Muawya, Paul, Salil, and Sandeep, for grilling me during my practice talks, and more importantly, keeping the lab humored. I wish them the very best in their doctoral pursuit. Thanks to Muawya for all the rides to the airport. I sincerely thank my former lab mates Ali, Aravindh, Ravi, Sailashri, Saurabh and Shobhit, for their friendship and support as a fellow grad student.

I owe a big thanks to Sandy Bronson for taking care of all my paperwork and conference reimbursements. I also want to thank Patrick Murphy for his condor support, and Aaron for his VCL support, which helped me meet various conference deadlines.

Finally, I would like to acknowledge the sponsors for my research. This research was supported by NSF CAREER grant No. CCR-0092832, and generous funding and equipment donations from Intel. Any opinions, findings, and conclusions or recommendations expressed in this dissertation are those of the author and do not necessarily reflect the views of the National Science Foundation.

TABLE OF CONTENTS

List of Tables viii
List of Figures ix
Chapter 1 : Introduction 1
1.1 Prediction-based Partial Redundant Threading (PRT) on Slipstream 4
1.2 A Canonical Fault-Tolerant Superscalar Processor 6
1.2.1 Canonical Fault Checks for a Superscalar Processor 8
1.2.2 Putting it All Together 11
Chapter 2 : Prediction-based Partial Redundant Threading 14
2.1 Fault Detection in Various PRT Models 17
2.2 Prediction-based PRT on Slipstream 23
2.2.1 Confident Branch Prediction 24
2.2.2 Confident Silent Write Prediction 24
2.2.3 Confident Dead Write Prediction 26
2.2.4 Confident Silent Store Prediction 28
2.2.5 Result: % Undetectable Faulty Instructions 28
2.3 Fault Recovery 28
2.3.1 Previous Recovery Implementation 29
2.3.2 Recovery Alterations 29
2.3.3 Direct Checks of Silent Writes and Stores 31
2.4 Novel Coverage Analysis 34
2.5 Simulation Environment 37
2.6 Results 39
2.6.1 Instruction Breakdown 39
2.6.2 Slipstream with Base Recovery 40
2.6.3 Slipstream with New Recovery 42
2.6.4 Performance 43
2.7 Using Only Branch Predictions 45
2.8 Summary 48
Chapter 3 : Inherent Time Redundancy 50
3.1 Characterizing Program Repetition in SPEC2K Benchmarks 51
3.2 ITR Cache 55
3.3 Microarchitecture Support for ITR 55
3.3.1 ITR Signature Generation 56
3.3.2 ITR ROB and ITR cache 57
3.3.3 Fault Detection and Recovery Coverage 59
3.3.4 Faults on ITR Components 60
3.3.5 Faults on the Program Counter (PC) 61
3.4 The ITR Cache Design Space 62
3.5 Fault Injection Experiments 66
3.6 Area and Power Comparisons 70

3.7 Summary 73
Chapter 4 : Register Name Authentication 75
4.1 RNA Previous Mapping Check 76
4.2 RNA Writeback State Check 79
4.3 Source Register Renaming Faults 82
4.4 Fault Injection 82
4.4.1 Experimental Setup 82
4.4.2 Results 84
4.5 Summary 89
Chapter 5 : Timestamp-based Assertion Checking 90
5.1 TAC Mechanism 91
5.2 Handling Wraparound of Timestamp Counter 94
5.3 Fault Injection 95
5.3.1 Experimental Setup 95
5.3.2 Results 97
5.4 Summary 98
Chapter 6 : A Canonical Fault-Tolerant Superscalar Processor 100
6.1 Fault Injection Strategy 100
6.2 Fault Analysis of a Superscalar Pipeline 103
6.2.1 Fetch and Decode Stages 105
6.2.2 Register Rename Stage 110
6.2.3 Dispatch Stage 111
6.2.4 Backend Stages 113
6.3 Fault-Checking Regimen 119
6.4 Experimental Methodology 121
6.5 Results 124
Chapter 7 : Related Work 137
7.1 Space Redundancy Based Approaches 137
7.2 Time Redundancy Based Approaches 140
7.3 Information Redundancy Based Approaches 144
7.4 Logic-Circuit-Level Approaches 146
Chapter 8 : Summary 147
Bibliography 149

LIST OF TABLES

Table 2-1. SMT microarchitecture for all three modes.. 38
Table 3-1. Number of static traces for SPEC.. 64
Table 3-2. Microarchitecture configuration.. 66
Table 3-3. List of decode signals. 67
Table 4-1. Microarchitecture configuration.. 83
Table 5-1. Microarchitecture configuration.. 96
Table 6-1. Microarchitecture configuration. 122

LIST OF FIGURES

Figure 2-1. Partial Redundant Threading (PRT) spectrum. ... 15
Figure 2-2. Full redundant execution. ... 18
Figure 2-3. Conventional PRT. ... 19
Figure 2-4. Prediction-based PRT ... 20
Figure 2-5 (a) and (b). Relaxing the checking constraints in prediction-based PRT. ... 21
Figure 2-6. Prediction-based PRT with indirect checking through confident predictions. ... 22
Figure 2-7. Confidently predicted branch removed from partial thread. ... 24
Figure 2-8. Confidently predicted silent write removed from partial thread. ... 26
Figure 2-9. Confidently predicted dead write removed from partial thread. ... 27
Figure 2-10. Forward slice example. ... 37
Figure 2-11. Breakdown of retired instructions. ... 39
Figure 2-12. Coverages of Slipstream with base recovery. ... 40
Figure 2-13. Coverage of Slip+SS+SW_ORT_ret for various recovery approaches.. ... 43
Figure 2-14. Performance comparisons. ... 44
Figure 2-15. Coverage for Single+B(+b). ... 46
Figure 2-16. Optimistic coverage for Single+B(+b). ... 47
Figure 2-17. Performance of Single+B(+b). ... 48
Figure 3-1. Dynamic instructions per 100 static traces (integer benchmarks). ... 53
Figure 3-2. Dynamic instructions per 50 static traces (floating point benchmarks). ... 53
Figure 3-3. Distance between trace repetitions (integer benchmarks). ... 54
Figure 3-4. Distance between trace repetitions (floating point benchmarks). ... 54
Figure 3-5. Superscalar processor augmented with ITR support. ... 58
Figure 3-6. Loss in fault detection coverage. ... 66
Figure 3-7. Loss in fault recovery coverage. ... 66
Figure 3-8. Fault injection results. ... 69
Figure 3-9. Die photo of the IBM S/390 G5 processor (Source: [51]). ... 71
Figure 3-10. Energy of ITR cache vs. I-cache. ... 73
Figure 4-1. Fault detection using RNA's "previous mapping check". ... 78
Figure 4-2. Fault detection using RNA-writeback. ... 81
Figure 4-3. Breakdown of outcomes of RNA fault injection campaign. ... 85
Figure 4-4. Relation of faults to outcomes. ... 86
Figure 4-5. Relation of faults to assertion checks. ... 88
Figure 5-1. Example data-flow graph with latencies. ... 91
Figure 5-2. Illustration of TAC checks. ... 93
Figure 5-3. Breakdown of outcomes of TAC fault injection campaign. ... 98
Figure 6-1. Superscalar processor pipeline for (a) non-memory and (b) memory instructions. ... 104
Figure 6-2. Fault analysis of the fetch and decode stages of the pipeline. ... 109
Figure 6-3. Fault analysis of the register rename stage of the pipeline. ... 111
Figure 6-4. Fault analysis of the dispatch stage of the pipeline. ... 113

Figure 6-5. Fault analysis of the backend of the pipeline (Issue, Reg Read, Execute/Agen, Writeback, Cache Access, and Retire).. 118
Figure 6-6. Classification chart for fault injection outcomes. 124
Figure 6-7. Distribution of faults injected across different pipeline stages. 125
Figure 6-8. Fault outcome distribution. 126
Figure 6-9. Fault outcome distribution per pipeline stage. 127
Figure 6-10. Fault outcome distribution for each fault type injected in the fetch stage. 132
Figure 6-11. Fault check distribution for fetch stage. 132
Figure 6-12. Fault outcome distribution for each fault type injected in the decode stage. ... 133
Figure 6-13. Fault check distribution for decode stage.. 133
Figure 6-14. Fault outcome distribution for each fault type injected in the rename stage. .. 134
Figure 6-15. Fault check distribution for rename stage. 134
Figure 6-16. Fault outcome distribution for each fault type injected in the dispatch stage.. 135
Figure 6-17. Fault check distribution for dispatch stage.. 135
Figure 6-18. Fault outcome distribution for each fault type injected in the backend stages. 136
Figure 6-19. Fault check distribution for backend stages. 136

Chapter 1: Introduction

Processor performance has increased significantly over the years, fueled by deep technology scaling. Unfortunately, deep scaling also increases vulnerability to soft errors caused by cosmic rays in the atmosphere, most notably neutron particles. Several research studies [34][45][46][87][88][89][90] and practical reports [47][48][49][50] have shown that neutron particles striking silicon chips at sea level can induce sufficient charge to flip the logical state of memory elements (logic state bits or SRAM cells), or cause transient changes to outputs of combinational logic, which might get erroneously latched. Process technology scaling leads to scaling down of many important parameters that increase the vulnerability of a node from neutron strikes. Most importantly, when a process technology is scaled, the critical charge needed to change the logical state of a node (known as Q_{crit}) experiences a quadratic decrease [46], making it easier to flip a bit. In contrast, a quadratic decrease is observed in the charge collection area of a node [46], reducing vulnerability to soft errors. However, as each new process leads to more nodes on a chip (Moore's law), the net impact is expected to be a rise in overall chip soft error rates. Studies indicate that the soft error rate (SER) of logic state bits will rise 8% for each new generation [46], and that of combinational logic will increase several orders of magnitude across generations [34]. One study expects the overall chip SER at 16 nm to be 100 times that at 180 nm [45].

Given this trend, transient fault tolerance is becoming an important challenge in high performance microprocessor design. Efficient fault tolerance (in terms of power, area and

performance) is equally important because heavy-weight solutions will offset the benefits of high-performance processors, and hence slow down growth that the processor industry has seen over the years.

Conventional fault tolerance solutions confirm correctness by either duplicating explicitly in time or space, or adding information redundancy like parity, ECC, etc. Such traditional techniques incur high overheads that are unsuitable for commodity high-performance processors. For example, redundant multithreading [19][20][23][27][29] is a time-redundant execution model where all instructions of a program are duplicated, and their outcomes compared, to confirm correctness. Redundant multithreading has high performance and power overheads due to full-program duplication. Space redundancy also incurs high power and area overheads. A classic example of space redundancy is the IBM S/390 mainframe processor [51][52], in which the fetch, decode and execution units are duplicated, and signals from duplicated units are compared to detect transient faults. Information redundancy techniques like parity and ECC require checking circuitry that can impact cycle time if the protected elements lie on critical paths of the processor. Moreover, they have limited applicability in protecting combinational logic elements.

In this dissertation, a new approach is presented to achieve efficient fault tolerance based on microarchitecture insights. The idea is to harness microarchitecture knowledge to incorporate fault checks into existing microarchitecture mechanisms. Fault checks at the microarchitecture level have a unique advantage over other solutions in lowering the overheads of fault tolerance. Since they establish correctness of the microarchitecture, they provide broad fault coverage to many logic blocks with a few checks. Microarchitecture-

level fault checks are an effective middle-ground between architecture-level checks like redundant multithreading, which have high overheads due to duplication, and logic-circuit-level checks like parity, robust flip-flop designs [100][103][104], self-checking state machines [106][105][107] etc., which are fine-grain and lead to high overheads when applied to all logic.

In this dissertation, two fault-tolerant architectures are presented that use microarchitecture insights for fault tolerance.

The first architecture improvises an existing leader-follower multithreading architecture called Slipstream [16][27] to make it robust, using the insight that a correct prediction of an instruction's outcome is the same as the outcome produced by fault-free execution of the instruction. So, an instruction with a predictable outcome is not duplicated, rather, its outcome is compared to a prediction, with a misprediction being a strong indicator of a transient fault. An incorrect prediction can mask a faulty outcome, if they match. The vulnerability due to mispredictions can be reduced using confidence counters to guide the selection of predictions that are reliable proxies for redundant execution of instructions. The idea of using confident predictions as effective proxies for redundant execution forms the basis of my first thesis about **prediction-based partial redundant threading (PRT)**. *My first thesis is that near-100% fault coverage can be achieved without full duplication, by using highly-confident predictions to replace redundant execution of instructions.*

In the second architecture, a regimen of fault checks is developed for out-of-order superscalar processors. The fault checks exploit microarchitecture insights about superscalar mechanisms and are generic enough to be applied to any out-of-order superscalar processor

design. This forms the basis for my second thesis about using **canonical fault checks for a fault-tolerant superscalar processor**. *My second thesis is that, by exploiting microarchitecture insights, there is potential to replace conventional time or space redundancy approaches with an inexpensive regimen of canonical fault checks that provide substantial fault coverage of the processor, leading to a canonical fault-tolerant superscalar processor.*

The two architectures are discussed in further detail in Sections 1.1 and 1.2.

1.1 Prediction-based Partial Redundant Threading (PRT) on Slipstream

The main idea behind prediction-based PRT is to leverage predictions as a source of redundancy, and use them as proxies for redundant execution of instructions. The very same notion is also exploited by Slipstream processors for high-performance reasons, and hence, Slipstream provides an ideal platform to test my first thesis about the efficient fault tolerance of prediction-based PRT.

Slipstream [16][27][6][24][17][15][13][18][14] employs two copies of the program in a leader/follower arrangement. In the first copy, called the A-stream, confident predictions replace highly predictable instructions and instructions feeding them. The shorter A-stream runs faster because it retires fewer instructions. However, it is a speculatively-constructed thread, and needs to be checked for correct execution. The follower thread, called the R-stream, serves this purpose. It fetches and executes all instructions and compares its outcomes to those of the A-stream. The R-stream is also accelerated in the process, because it executes free of all control-flow and many data-flow dependences by leveraging the A-

stream outcomes as near-ideal predictions. The net result is faster execution compared to conventional non-redundant execution.

An additional benefit of Slipstream is transient fault tolerance due to some amount of duplication of the program. Previous studies of Slipstream [27][16] concluded that "the system transparently recovers from transient faults affecting redundantly-executed instructions". By building upon the insight that confident predictions are as effective as actual outcomes, it will be shown in this dissertation that Slipstream processors inherently possess higher levels of fault tolerance than suggested in previous studies, and minor improvisations to the microarchitecture can boost Slipstream's fault tolerance to nearly 100%. An improvised Slipstream processor on a SMT substrate that has close-to single-thread performance and close-to 100% fault coverage will be demonstrated as proof of my first thesis.

Chapter 2 discusses prediction-based PRT in depth. A general discussion of various PRT models is followed up with a thorough discussion of prediction-based PRT on Slipstream. Based on analysis of the various prediction scenarios, it is shown for the first time that Slipstream has high fault detection coverage of instructions (99.9%), as opposed to lower estimates from previous studies that only claimed coverage for duplicated instructions. It is shown that Slipstream's existing recovery implementation fails to capitalize on the excellent fault detection coverage. A suite of microarchitecture alterations are proposed to enhance Slipstream's fault recovery coverage. Slipstream's fault recovery coverage of instructions depends on how far the program is rolled back on a fault (the farther the rollback, the higher the recovery coverage). A forward-slice algorithm is developed to

estimate the fault recovery coverage for various recovery implementations. The forward-slice algorithm is then applied to estimate fault recovery coverage for a superscalar architecture that rolls back to the head of the reorder buffer on branch mispredictions.

1.2 A Canonical Fault-Tolerant Superscalar Processor

Traditional transient fault tolerance approaches that use time or space redundancy provide robust fault tolerance, but have significant overheads. The following drawbacks associated with these approaches lead to consideration of alternative fault-tolerant architectures:

1. Power: Power consumption has become a major consideration for high-performance processors. Time or space redundancy based implementations that protect the entire processor pipeline have substantial power overheads, making them unattractive for high-performance processors [74][75].
2. Complexity: Though time or space redundancy architectures seem quite straightforward, there could be significant implementation complexities. The problem stems from the fact that the two threads of execution must be synchronized for result correlation. Lockstepped execution used in space redundancy approaches [40][51], has significant pin-interface complexities if the duplication is at the core level [40] or wiring complexities if the duplication is within the core [51]. Lockstepped execution also has significant implementation challenges in modern microprocessors [65]. Staggered execution used in redundant multithreading also has significant implementation complexities [9].

3. Performance: Time redundant execution incurs performance overheads because instructions compete for resources on a single processor core. For example, redundant multithreading [20][23] incurs high performance overheads that may make it unsuitable for high-performance processors.

4. Area: Straightforward duplication of resources could lead to significant area overheads. For example, in the IBM S/390 G5, the fetch, decode and execution units were duplicated, with a 20% to 30% area overhead [51].

Prediction-based PRT, which forms the first part of this dissertation, reduces the performance overhead of redundant multithreading architectures by reducing the number of duplicated instructions. However, not all programs are predictable, hence, opportunities for reducing duplication might be limited in these programs. Moreover, time redundancy models that exploit prediction-based PRT still have power and complexity drawbacks indicated above.

A more fundamental approach to make fault tolerance viable for commodity high-performance processors is to develop fault checks that can be incorporated into the processor microarchitecture without significant overheads. In this spirit, my dissertation proposes some unconventional fault checks based on microarchitecture insights, that provide significant fault protection in an efficient manner. An important characteristic of all the fault checks is that they are "canonical," i.e., they are generic enough to be applied to any out-of-order superscalar processor design.

The ultimate goal of this approach is to develop a canonical fault-tolerant superscalar processor by engaging a regimen of low-overhead microarchitecture-level fault checks. Each

check would protect a distinct part of the processor pipeline, thus, the regimen as a whole would provide comprehensive protection to the processor.

1.2.1 Canonical Fault Checks for a Superscalar Processor

Towards the goal of achieving a canonical fault-tolerant superscalar processor, my dissertation develops three important microarchitecture-level fault checks for out-of-order superscalar processors: Inherent Time Redundancy (ITR) for protecting decode signals, Register Name Authentication (RNA) for protecting register-dependency information, and Timestamp-based Assertion Checking (TAC) for ensuring correct issue ordering among instructions. The following subsections discuss the three checks.

1.2.1.1 Inherent Time Redundancy (ITR)

Inherent Time Redundancy (ITR) is based on the insight that many programs repeatedly execute the same instructions within close time periods. This program repetition can be exploited for fault tolerance by recording input-independent signals in the processor (like decode signals) and checking correctness upon program repetition by comparing the new instance of a signal with the recorded instance. An ITR cache is proposed for recording the decode signals and it is shown that a relatively small ITR cache can provide fault coverage on the decode signals for a significant fraction of instructions. By checking decode signals for correctness, ITR can potentially detect faults in the fetch unit, faults in the decode unit, and faults in all pipeline latches that store decode signals of instructions.

Chapter 3 discusses ITR in detail. It also discusses one possible implementation to exploit ITR for fault tolerance in a superscalar processor. Decode signals of instruction traces (where a trace is a sequence of instructions terminating either at a branch instruction or at the

end of 16 instructions) emerging from the decode stage are combined into a signature and stored in the ITR cache. Subsequently, newly formed decode signatures are checked against cached decode signatures to check for transient faults. Chapter 3 evaluates the fault tolerance capabilities of ITR using targeted fault injection on decode signals of a processor, and also compares the ITR cache's area and power overheads against conventional approaches that are based on space or time redundancy, highlighting the efficiency of ITR-based approaches.

1.2.1.2 Register Name Authentication (RNA)

Register Name Authentication (RNA) exploits redundancy among renaming structures used in out-of-order superscalar processors, to detect faults in the destination register mappings of instructions. In particular, RNA includes two checks, the RNA previous mapping check and the RNA writeback state check. The RNA previous mapping check is based on the insight that, when an instruction's logical destination register is renamed, the previous physical register mapping in the rename map table corresponds to the previous producer of that logical destination register, and the same previous physical register mapping should be in the architectural map table when the instruction commits. The first RNA check involves comparing the previous physical register mapping recorded at the register renaming stage to the corresponding mapping in the architectural map table at the commit stage. The first RNA check can detect faults in several rename structures: the rename map table, architecture map table, shadow map tables (branch checkpoints), and all pipeline latches that store destination physical register mappings including some fields of the ROB. However, the first RNA check cannot detect faults in which an erroneous mapping appears consistent among all the structures. For example, faults in the free list and the destination renaming

logic will cause an erroneous mapping very early in the pipeline, and all the renaming structures are updated consistently (rename map, architectural map, shadow maps, and ROB).

The RNA writeback state check aims to detect such faults using the insight that they cause register conflicts between the instructions. Basically, an erroneously mapped physical register might already be in use by another active instruction, committed to architectural state, or still available in the free list. Asserting the state of a physical register (whether it is in the free list, whether it is not ready, etc.) at key points in the processor pipeline (e.g., the register writeback stage) exposes this conflict, hence, detects the fault.

Chapter 4 discusses RNA in depth, illustrating the various faults involving destination physical register mappings, and discussing how the two RNA checks offer protection against such faults. Chapter 4 also evaluates the fault tolerance capabilities of RNA using targeted fault injection on structures associated with register renaming in an out-of-order superscalar processor.

1.2.1.3 Timestamp-based Assertion Checking (TAC)

Timestamp-based Assertion Checking (TAC) exploits the insight that an instruction should execute only after all its producers have executed. This invariant is true even in an out-of-order superscalar processor, where instructions that do not depend on each other issue in parallel or out of order. To capture this time-orderliness within a data dependence chain, TAC assigns timestamps to instructions when they issue, and asserts dependent instructions issued only after their producers. The out-of-order scheduler comprises of complex hardware structures that track data dependencies, and complex logic to wake up and select correct

instructions for issuing. A transient fault in the scheduler structures or any associated logic involved can cause instructions to misfire. TAC detects all of these faults with one simple assertion check.

Chapter 5 discusses TAC in detail. Microarchitecture alterations required to incorporate timestamp counters and the TAC checks are explained. Timestamp counter overflow is a problem, since it can cause false alarms. An elegant solution is proposed to handle overflows without compromising on fault detection. Finally, the fault tolerance capabilities of TAC are evaluated using targeted fault injection.

1.2.2 Putting it All Together

The culmination of this work is a fault-tolerant superscalar processor that incorporates all the fault checks developed in this dissertation. The goal is to make the case for a fault-tolerant processor based on a regimen of low-overhead fault checks, where each check targets a different part of the pipeline, and the regimen as a whole provides comprehensive coverage of the pipeline.

Chapter 6 builds a checking regimen that includes all the fault checks developed in this dissertation: ITR, RNA, TAC and mispredictions of confident branches. Some fault checks are improvised from their original description in earlier chapters and new fault checks are introduced for broader coverage of the processor pipeline.

To test the fault tolerance of a processor augmented with the checking regimen, randomized fault injection is carried out across the processor pipeline in a timing simulator. All faults are injected on latches (faults on combinational logic are also modeled as faults on output latches). The source of a fault is a bit-flip of a combinational or sequential logic

element. A straightforward approach to inject faults would be to consider all pipeline latches as candidates for fault injection, and flip a bit of a randomly selected latch. However, this direct approach is not feasible because a timing simulator does not model all the pipeline latches.

My dissertation presents a practical fault injection approach for a timing simulator of a superscalar processor. The idea is to aggregate faults at several locations in a pipeline stage into visible fault manifestations that have a greater chance of causing a failure. In reality, faults in many pipeline latches get masked away and do not become errors. Aggregating several faults into visible fault locations increases the likelihood of an error due to a fault, and to a certain degree, reduces masking due to dormant faults from the experiments. Modeling dormant faults would only highlight the presence of fault masking in the processor. Hence, the approach of aggregating faults into visible fault locations by inspection provides a rigorous evaluation of processor fault-detection capabilities, while not sacrificing pipeline coverage in the fault injection experiments.

In Chapter 6, comprehensive analysis is done on all pipeline stages of a superscalar processor to aggregate faults in each pipeline stage into visible fault manifestations. A detailed superscalar pipeline is presented for analysis, mostly based on the Illinois Verilog Model (IVM), an open-source verilog description of an Alpha microprocessor from Prof. Sanjay Patel's research group at the University of Illinois [61]. The outcome of the comprehensive analysis is a list of faults for each pipeline stage, which are modeled and randomly triggered in the timing simulator.

As proof of my second thesis, it will be shown that incorporating the checking regimen, comprising of the critical fault checks introduced in my dissertation, into a superscalar processor pipeline leads to substantial fault coverage, making the case for a canonical fault-tolerant superscalar processor.

Chapter 2: Prediction-based Partial Redundant Threading

The microarchitecture insight behind prediction-based PRT is that highly confident predictions are effective proxies for redundant execution. The reasoning is that a correct prediction for an instruction is as good as the actual outcome achieved by executing the instruction. On the other hand, a misprediction of an instruction outcome can mask a fault if the mispredicted outcome matches the faulty outcome. Since highly confident predictions are rarely incorrect, confident predictions can be used as effective proxies for redundant execution of instructions.

In redundant multithreading architectures [19][20][23][27][29], all instructions are fetched and executed twice, via two redundant threads on a simultaneous multithreading (SMT) pipeline. The results of the duplicated instructions are compared to detect transient faults. The performance overhead of dual redundant threads on an SMT pipeline can be significant due to resource contention (fetch, issue, and retire bandwidth, physical registers, etc.) and checking bandwidth. Lighter weight approaches have been proposed [5][27][30], that duplicate only a subset of the dynamic instruction stream. These approaches are referred to as Partial Redundant Threading (PRT) and will be discussed in the context of the PRT spectrum shown in Figure 2-1.

- *Partial duplication*. In this approach, instructions are arbitrarily duplicated for fault tolerance. For example, Opportunistic Fault Tolerance [5] duplicates instructions only during periods of poor single-thread performance (e.g., during L2 cache misses or

low instruction-level parallelism). This approach only achieves partial fault tolerance depending on the amount of duplication.

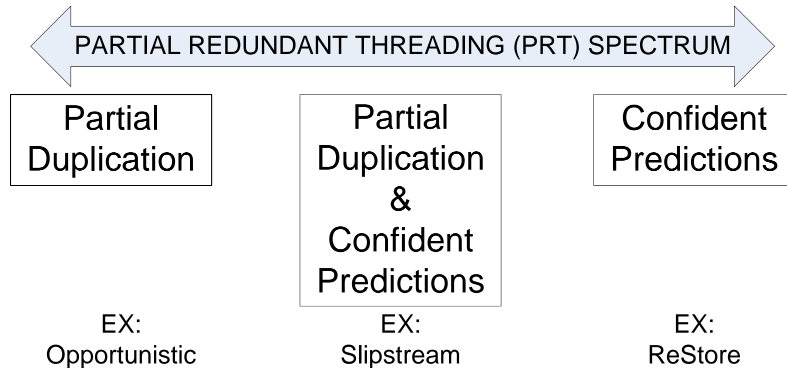


Figure 2-1. Partial Redundant Threading (PRT) spectrum.

- *Confident predictions*. This approach is an extreme case of prediction-based PRT where only confident predictions provide redundancy to detect faults. Since highly confident predictions are rarely incorrect, a misprediction most likely indicates that a fault afflicted the instruction outcome, and not that the prediction is wrong. For example, ReStore [30] does not explicitly duplicate instructions. Instead, highly confident branch predictions are used to detect faults. Whether due to a fault or a misprediction, the processor rolls back to a prior register/memory checkpoint. If there was a fault, rolling back masks the fault if the original faulty instruction is logically after the checkpoint (i.e., not yet retired) and therefore the checkpoint is not corrupted. Since the number of confident predictions in a program is limited, only partial fault tolerance is achieved by this prediction-based PRT approach.

- *Partial duplication and confident predictions*. This approach is a more complete case of prediction-based PRT as both confident predictions and partial duplication are combined to provide redundancy. For example, Slipstream [16][27] combines partial

duplication and confident predictions. A second reduced thread is created by (i) removing predictable branches and their backward slices, replacing them with highly confident branch predictions, (ii) removing predictable dynamically-dead instructions and their backward slices, and (iii) removing predictable silent writes (they do not change the value in a location) and their backward slices, implicitly replacing them with highly confident value predictions. This form of prediction-based PRT holds promise for near complete fault tolerance as regions not protected by highly confident predictions are protected through duplication.

To reiterate from Chapter 1, my first thesis is that if a very small sacrifice in fault coverage is acceptable (in the order of 0.01% of instructions), then, near-100% fault coverage can be achieved without full duplication by using highly confident predictions to replace redundant execution of instructions. Such prediction-based partial redundant threading (PRT) will pave path to achieving close-to single-thread performance with fault coverage close to that of full duplication, a highly desirable goal for reliable high-performance processors. In this chapter, Slipstream will be used as a platform to study and prove my thesis.

To prove my thesis, it will be shown for the first time that the mixture of partial duplication and confident predictions in Slipstream actually closely approximates the coverage of full duplication. From a performance perspective, Slipstream significantly reduces the amount of duplication (as much as 57% of instructions are covered by confident predictions alone), and it will be shown that it can achieve performance close to that of single-thread execution, and is far more efficient than full redundant execution.

The rest of this chapter is organized as follows. Section 2.1 generically discusses fault detection and recovery in various PRT models. Section 2.2 discusses prediction-based PRT in Slipstream and thoroughly dissects the four prediction scenarios in Slipstream, illuminating cases in which faulty instructions are detectable vs. undetectable. It is shown for the first time, that Slipstream is able to detect faults in nearly 100% of dynamic instructions. Section 2.3 discusses fault recovery in Slipstream, highlighting existing weaknesses that cause it to fall far short of 100% instruction coverage despite excellent detection capability. Several recovery alterations are suggested in Section 2.3.2 to increase Slipstream's fault recovery coverage. Section 2.4 discusses a novel analysis framework to measure fault coverage of various recovery implementations. Section 2.5 discusses the simulation environment. Section 2.6 discusses fault coverage and performance results of Slipstream with various recovery alterations. Section 2.7 discusses fault coverage results for a prediction-based PRT approach that only uses confident branch predictions to detect faults. Finally, Section 2.8 summarizes and concludes the chapter.

2.1 Fault Detection in Various PRT Models

In full redundant multithreading [19][20][23][27][29], all instructions are duplicated and their results cross-checked to detect transient faults. For example, in Figure 2-2, f1 and f2 refer to redundant threads. The instructions A and B are duplicated and their results compared to detect transient faults. Since a fault inflicting either instruction A(f1) or B(f1) will be detected by their counterparts A(f2) or B(f2) respectively (shown by tick marks next to the instructions), the fault detection coverage is 100%. Also, since fault detection will occur at the source faulty instruction, the fault is recoverable by restarting both threads from

the instruction that detected the fault, and fault recovery coverage is also 100%. However, the main drawback of full redundant multithreading is the significant performance degradation due to heavy resource contention between instructions of the fully duplicated redundant threads.

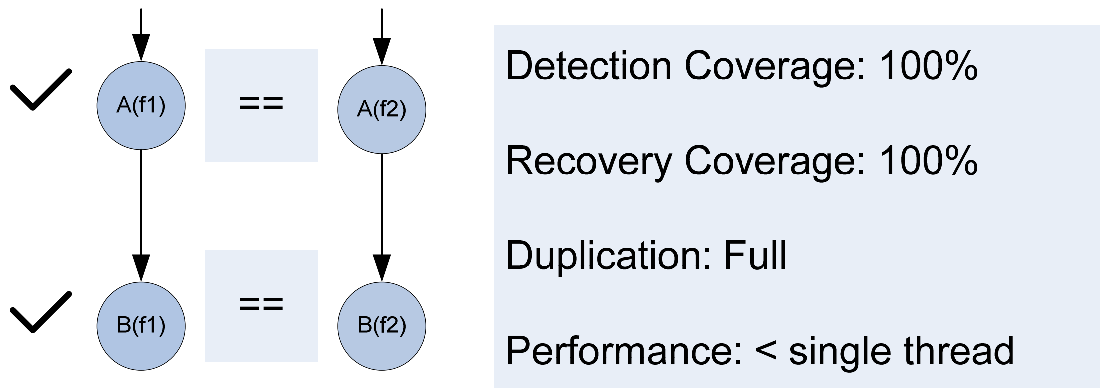


Figure 2-2. Full redundant execution.

In conventional PRT, the partial redundant thread is constructed by arbitrarily duplicating instructions from the full thread. For example, in Figure 2-3, instructions B(f) and C(f) have redundant counterparts B(p) and C(p) in the partial redundant thread, whereas, instruction A(f) is not duplicated. In conventional PRT, the instructions missing in the partial redundant thread are emulated by copying their effects from the full thread. Due to the dependence of the partial redundant thread on the full thread, only faults affecting duplicated instructions can be detected. Non-duplicated instructions represent a common mode failure, i.e., a fault on a missing instruction (like A(f) in Figure 2-3) creates the same faulty effect in the both threads, and hence, goes undetected. The fault recovery coverage of conventional PRT is same as the fault detection coverage, since fault detection occurs at the source of the fault. Conventional PRT approaches incur much lower performance overhead than full duplication. For example, in opportunistic fault tolerance, instructions are duplicated only

during L2 misses or low-ILP phases, and only an average 2% performance degradation was reported to be incurred across several SPEC2000 benchmarks [5]. The average redundancy obtained across SPEC2000 benchmarks was 60% [5], well short of complete fault coverage.

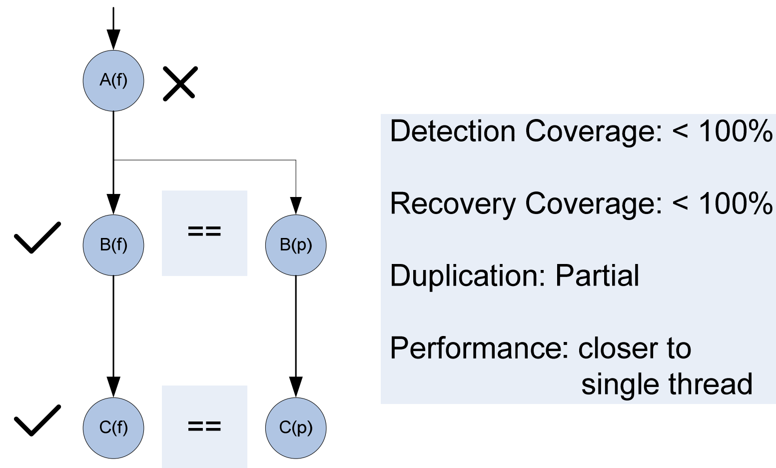


Figure 2-3. Conventional PRT.

In prediction-based PRT, the amount of instruction duplication is reduced by replacing highly predicable instructions in the redundant thread with confident predictions. The insight is that a correctly predicted outcome is as good as the actual outcome generated by executing the instruction. An incorrectly predicted outcome can mask a fault if the faulty outcome matches the incorrect prediction. However, if the confidence level for a prediction is high, it's very likely to be correct and the vulnerability caused by mispredictions is minimized. For example in Figure 2-4, instruction A(f) is highly predictable and its confident prediction P_A replaces it in the partial redundant thread. A correct P_A can detect faults afflicting A(f) whereas an incorrect P_A can potentially mask a fault in A(f). By using predictions to replace instructions, the partial redundant thread is shortened in a self sufficient manner and does not need to copy state from the full thread. The fault detection and recovery coverage of the prediction-based PRT model shown in Figure 2-4 is close to

100% provided sufficiently high confidence is used. The loss in coverage is due to the rare mispredictions of confident predictions that act as proxy for the replaced instructions. From a performance standpoint, if only highly predictable instructions with confident predictions are removed from the partial thread, the reduction will not be sufficient enough for prediction-based PRT to match single thread performance, due to the limited supply of highly confident predictions in a program.

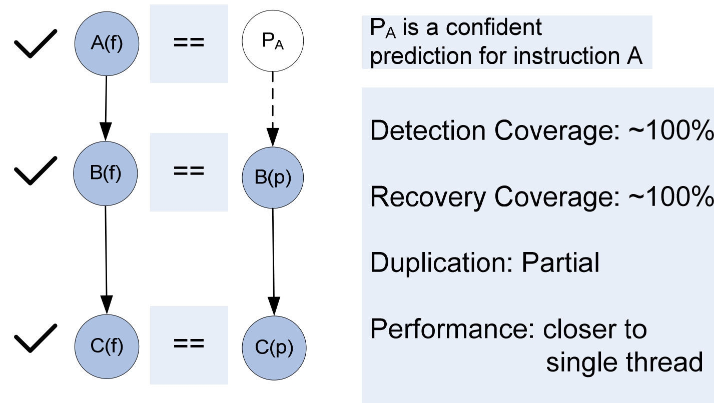


Figure 2-4. Prediction-based PRT.

To further shorten the partial thread in prediction-based PRT, it is observed that when a confident prediction replaces an instruction, apart from being a proxy to the outcome produced by that instruction, the prediction also eliminates a consumer for a prior instruction. This can lead to instructions in the partial thread whose consumers are all replaced by confident predictions. Such instructions can be removed since their values are not consumed by any other instructions. The only reason they have to be retained in the partial thread is to check their counterparts in the full thread for transient faults. A key observation that a confident prediction can detect faults in instructions even in its backward slice enables elision of this check and the removal of instructions that are dynamically dead because of

their consumers being replaced by confident predictions. This observation can lead to significant shortening of the partial redundant thread, and bring the performance of prediction-based PRT close to that of a single thread, while retaining near 100% fault coverage.

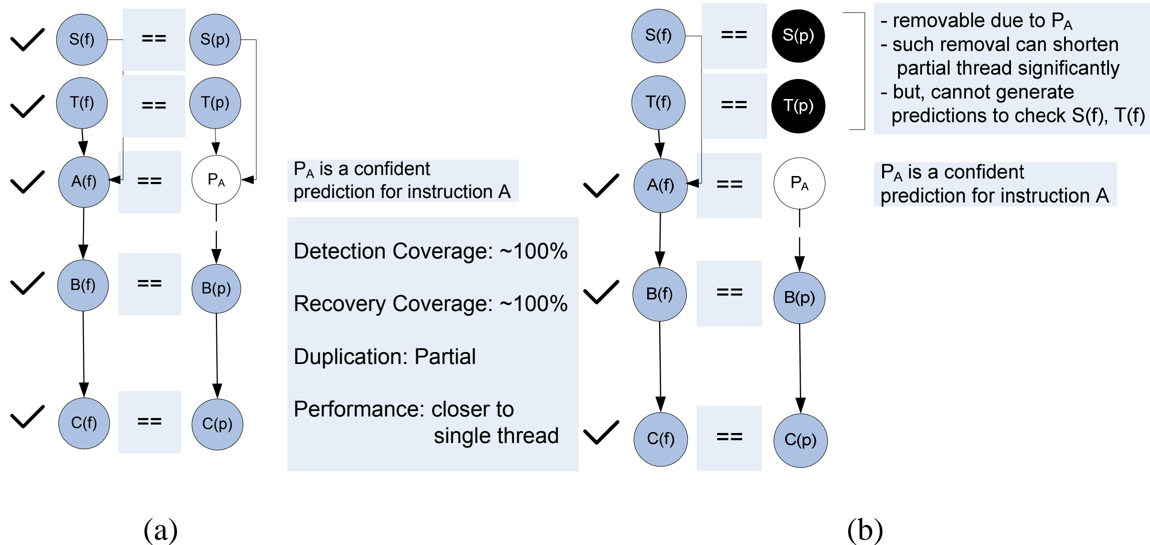


Figure 2-5 (a) and (b). Relaxing the checking constraints in prediction-based PRT.

Figure 2-5 depicts the above observations. As seen in Figure 2-5-a, the outcomes of instructions S and T are consumed by instruction A. Since the confident prediction P_A replaces A(p), the results of S(p) and T(p) will not be consumed by any instruction. Hence, S(p) and T(p) can be removed from the partial thread, as shown in Figure 2-5-b. However, doing so would leave their counterparts S(f) and T(f) in the full thread unchecked. This is because S and T are not highly predictable instructions and there are no confident predictions to replace them. The key observation that enables removing S(p) and T(p) from the partial thread is that S(f) and T(f) get checked indirectly by the prediction P_A, as shown in Figure 2.1.f. Removing instructions whose forward slice has confident predictions as leaves would lead to significant shortening of the partial thread, and bring the performance of prediction-

based PRT close to that of single thread execution, and maintaining the fault detection coverage close to 100%. The fault recovery coverage depends on how recovery is performed on fault detection. Conventional recovery where the threads are restarted from the source of the fault will lose coverage on non-duplicated instructions checked indirectly by confident predictions. Enhanced recovery models that rollback to a prior point would improve recovery coverage, and will be discussed later in the context of Slipstream processors.

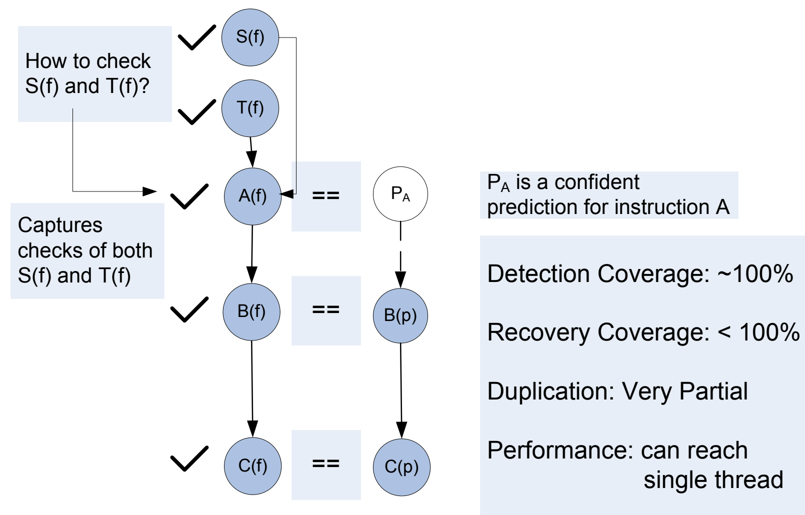


Figure 2-6. Prediction-based PRT with indirect checking through confident predictions.

To conclude this section, it was discussed why conventional PRT schemes only provide partial fault tolerance. Prediction-based PRT schemes have the capability to provide near-100% fault tolerance using confident predictions to replace instructions in the partial thread. Further removal of instructions whose forward slices have confident predictions as leaves can lead to significant reduction of the partial thread, and has the potential to bring the performance of prediction-based PRT close to that of a single thread with near-100% fault detection coverage. However, conventional recovery from the source of the fault does not

provide good coverage and enhanced recovery models that rollback to an earlier point would be needed to capitalize on the near-100% fault detection coverage.

2.2 Prediction-based PRT on Slipstream

So far prediction-based PRT has been discussed generically. Slipstream specifically exploits four types of prediction to form the partial thread.

- *Branch prediction*. Highly predictable branches and their backward slices are removed.
- *Silent write prediction*. This is a special case of value prediction. Instructions and their backward slices are removed, that predictably write the same value into a logical register as the previous write to the logical register. The implied value prediction is the value in the logical register prior to writing it.
- *Dead write prediction*. Predictably dead register-writing instructions and their backward slices are removed.
- *Silent store prediction*. Store instructions and their backward slices are removed, that predictably store the same value into a memory location as the previous store to the location.

In the following subsections, each of the four prediction scenarios are analyzed in depth to understand when faults are detectable vs. undetectable. The key idea is to consider both correct prediction and incorrect prediction. The analysis reveals that faulty instructions are undetectable only when faults coincide with mispredictions. Fortunately, mispredictions among confident predictions are very rare.

2.2.1 Confident Branch Prediction

Figure 2-7 shows a confidently predicted branch I and its producer H. Annotations (f) and (p) denote which thread an instruction belongs to, full or partial, respectively. The full thread has instances H(f) and I(f). Counterparts H(p) and I(p) are removed from the partial thread, as indicated by dashed circles and arcs, and replaced with a confident branch prediction. This confident branch prediction becomes the redundant counterpart of I(f). I(f) produces the wrong branch outcome due to a transient fault in either H(f) or I(f), depicted by an X over I(f). If the confident branch prediction is correct, then it will differ from the incorrect branch outcome of I(f) as shown in Figure 2-7(a), thus detecting a faulty H(f) or I(f). However, if the confident branch prediction is incorrect, depicted by an X over I(p), then it will match the incorrect branch outcome of I(f) as shown in Figure 2-7(b), failing to detect a faulty H(f) or I(f).

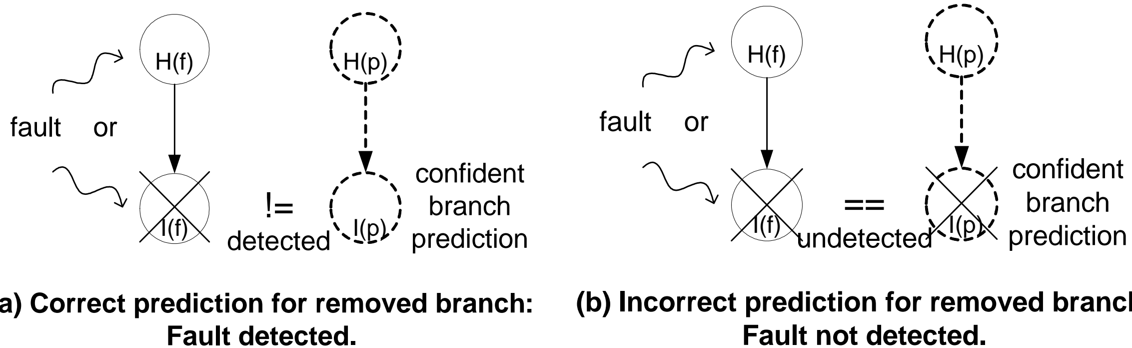
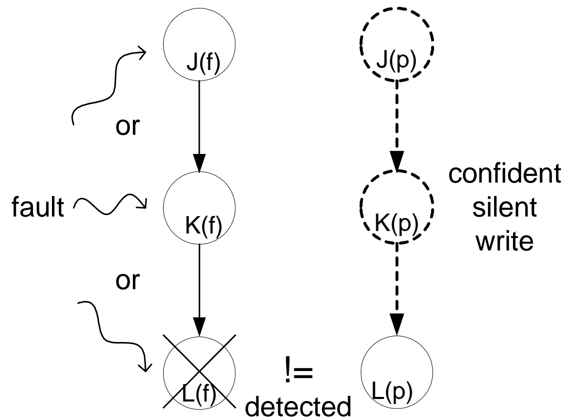


Figure 2-7. Confidently predicted branch removed from partial thread.

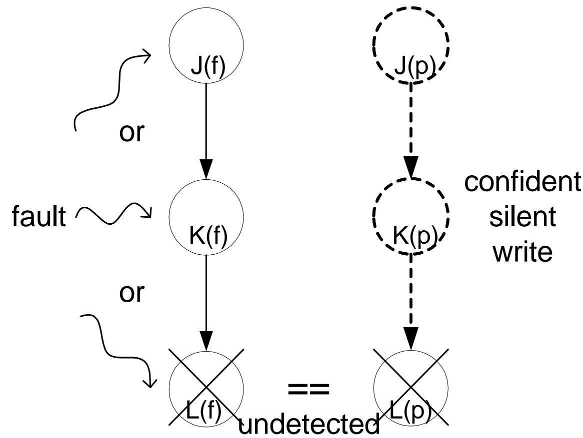
2.2.2 Confident Silent Write Prediction

Figure 2-8 shows a confidently predicted silent write K, and its producer J and consumer L. The full thread has instances J(f), K(f), and L(f). The predicted silent write K(p) and its producer J(p) are removed from the partial thread, as indicated by their dashed circles and arcs. They are implicitly replaced with a confident value prediction that is the value

produced by the last writer of K's logical destination register. That is, the consumer of the predicted silent write, L(p), remains in the partial thread and it is predictively renamed to the previous writer of K's register (not shown) instead of K(p) itself. As such, L(p) becomes a predictive redundant counterpart of L(f). L(f) produces a wrong result due to a transient fault in either J(f), K(f), or L(f), depicted by an X over L(f). If K(p) is truly a silent write, then L(p) produces the correct result, thus detecting a faulty J(f), K(f), or L(f), as shown in Figure 2-8(a). However, if K(p) is not actually a silent write, then L(p) may produce an incorrect result since it uses a stale value instead of the present value from K(p). If L(p) and L(f) are incorrect in exactly the same way, a faulty J(f), K(f), or L(f) is not detected, as shown in Figure 2-8(b).



(a) Predicted silent write correct:
Fault detected.

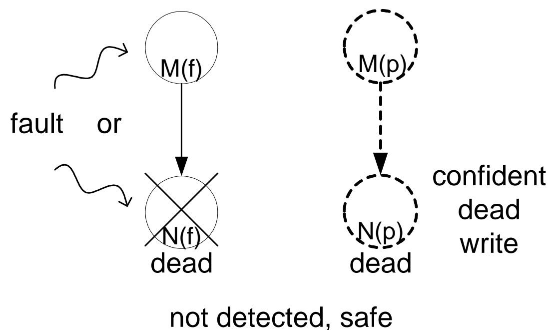


(b) Predicted silent write incorrect:
Fault not detected.

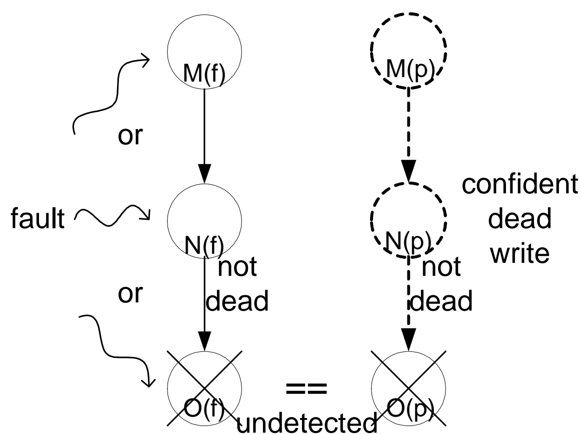
Figure 2-8. Confidently predicted silent write removed from partial thread.

2.2.3 Confident Dead Write Prediction

Figure 2-9 shows a confidently predicted dead write N and its producer M. The full thread has instances M(f) and N(f). The predicted dead write N(p) and its producer M(p) are removed from the partial thread, as indicated by their dashed circles and arcs. N(f) produces a wrong result due to a transient fault in either M(f) or N(f), depicted by an X over N(f). If N is truly dead as in Figure 2-9(a), then a faulty M(f) or N(f) will not be detected, because there



(a) Predicted dead write correct:
Fault not detected but safe.



(b) Predicted dead write incorrect:
Fault not detected.

Figure 2-9. Confidently predicted dead write removed from partial thread.

is no redundant counterpart for N(f) to compare against nor is there a redundantly executed consumer of N(f) to do an indirect comparison. However, an undetected faulty M(f) or N(f) is safe for this very reason – it is not referenced in the future. If N is actually live as in Figure 2-9(b), then an unforeseen consumer O is brought into the picture. O(f) produces a wrong result due to a transient fault in either M(f), N(f), or O(f), depicted by an X over O(f). The partial thread's counterpart, O(p), may also produce a wrong result because it consumes a stale value, instead of the value that was supposed to be produced by the removed, presumed

dead write N(p). If O(f) and O(p) produce the same wrong result, then a faulty M(f), N(f), or O(f) is not detected, as shown in Figure 2-9(b).

2.2.4 Confident Silent Store Prediction

The analysis for silent store predictions mirrors that of silent write predictions in Section 2.2.2 and Figure 2-8, where instruction K is a store and instruction L is a dependent load instruction. However, the store K(f) may be faulty in two ways: faulty value or faulty address. Figure 2-8 only shows the faulty value case. Nonetheless, a faulty store address merely causes the store K(f) to store to a different address than the load L(f), leading to the same situation of a faulty load L(f).

2.2.5 Result: % Undetectable Faulty Instructions

Since instruction-removal mispredictions are rare in Slipstream (due to conservative confidence), very few mispredictions are expected and correspondingly low loss in coverage. In all benchmarks studied, losses in coverage was measured to be below 0.1%. Therefore, Slipstream can detect faults in 99.9% of instructions.

Fault recovery is a separate issue. A good recovery implementation will capitalize as much as possible on the fault detection capability. Recovery is discussed in the next section.

2.3 Fault Recovery

Slipstream's fault recovery capability is next discussed, given its excellent fault detection capability. Section 2.3.1 reviews the previous recovery implementation and discusses its weaknesses. Section 2.3.2 proposes recovery improvements. Section 2.3.3 proposes early (direct) checks of silent writes/stores to improve the chances for successful (i.e., timely) recovery.

2.3.1 Previous Recovery Implementation

The Slipstream recovery implementation described in previous papers [16][27] works as follows. When a duplicated instruction detects a fault, it posts a fault exception and the Slipstream processor waits for the exception to reach the head of the reorder buffer (ROB). Then, the full and partial threads are re-synchronized and restarted from the faulty duplicated instruction. This recovery implementation attributes the fault to the duplicated instruction. However, it may be that its non-duplicated producer is faulty. Suppose this is the case. The above delayed recovery model permits the faulty producer to retire, corrupting the full thread's architectural state which is supposed to be a safe checkpoint for recovery. When recovery is initiated, the partial thread's architectural state inherits the flawed full thread's architectural state. Now, both threads are architecturally corrupted and the system is potentially unrecoverable, depending on whether or not the corrupt state is architecturally masked in the future.

2.3.2 Recovery Alterations

2.3.2.1 ROB-head Recovery

The problem with the previous recovery implementation is not so much delaying recovery until the detected fault reaches the head of the ROB, as restarting the threads from the faulty consumer instead of from the faulty producer. In other words, previous Slipstream implements "consumer recovery" instead of "producer recovery".

Producer recovery is more conservative since it always attributes a fault to the producer. However, the overhead of extra squashed instructions is a small price to pay for recovery from faults in non-duplicated instructions, the primary objective of this chapter. In

practice the overhead is negligible, since Slipstream mispredictions are rare thus recovery is rarely initiated.

Implementing producer recovery explicitly would require dependence vectors, so that a faulty consumer could mark its direct/indirect producers as potentially faulty. Instead, a hardware-free approach is proposed that emulates producer recovery (achieves the same coverage). Namely, the two threads are restarted from the oldest instruction in the ROB (ROB head) at the time the fault is detected. Recovery succeeds if the original faulty instruction is still in the ROB when the fault is detected, since it is prevented from retiring in this case.

2.3.2.2 ROB Occupancy Threshold

Slipstream staggers its two threads. Staggering enables the trailing thread to use outcomes from the leading thread as mostly perfect branch and value predictions [20][23]. Breaking dependences in the trailing thread causes it to fetch and execute more efficiently. In an SMT implementation, this releases resources back to the leading thread and thereby reduces overall execution time for the dual threads.

In Slipstream, efficiency means low occupancy of the ROB by the full thread (the trailing thread). While this is good for performance, it reduces the effectiveness of ROB-head Recovery. Low ROB occupancy means a faulty non-duplicated instruction is more likely to retire before it is detected by a consumer. ROB-head Recovery is unsuccessful in this case.

One countermeasure for boosting ROB occupancy is to target a ROB occupancy threshold (e.g., 16 or 32 instructions). The target is met and maintained by throttling full

thread retirement. The performance/coverage trade-off of various ROB occupancy thresholds is explored in the results section.

2.3.2.3 History Buffer

One way to increase the success rate of producer recovery without delaying retirement is to use a history buffer [25]. Before an instruction retires from the ROB and commits its value to the architectural register/memory state, the previous committed value is read and stored in the history buffer. When a fault is detected, the ROB is flushed as before (ROB-head Recovery). In addition, the architectural state is restored to an even earlier precise state by stepping through the history buffer in reverse and using the saved values to undo changes to the architectural state. If the original faulty instruction had retired from the ROB but not the history buffer, then the faulty committed value is safely removed.

The history buffer is a potentially simpler alternative than register and memory checkpoints advocated by ReStore but the principle is the same.

2.3.3 Direct Checks of Silent Writes and Stores

Since silent writes and stores are removed from the partial thread, their counterparts in the full thread have nothing to compare with. Therefore, faults affecting these instructions in the full thread are detected by future duplicated consumers (e.g., instruction L in Figure 2-8-a). An indirect check is not useful if it is too late to prevent retiring the original faulty instruction.

Fortunately, direct checks are possible for silent writes and stores. First, the Slipstream components that facilitate instruction removal (IR-detector and IR-predictor) can be augmented to remember the reason for speculatively removing an instruction. Therefore,

predicted silent writes and stores can be explicitly marked in the full thread (e.g., K(f) in Figure 2-8 is marked for direct checking). Second, predicted silent writes and stores can be directly checked in the full thread by comparing the value being written or stored with the value already in the register or memory location. If the values differ, either the prediction is wrong (not truly silent) or the silent instruction is faulty. Either way recovery is needed (to repair the partial thread or to mask the fault, respectively).

2.3.3.1 Methods for Directly Checking Silent Writes

Directly checking a predicted silent write in the full thread requires obtaining the value of the previous write to the same logical register. Five new approaches are proposed and listed below. The fourth approach (ORT_ret) is advocated because it essentially comes for free in the existing Slipstream implementation and results show it increases coverage about as well as the other approaches.

All approaches except ORT_ret require knowing the previous physical register mapping of the logical register since it is this physical register that will have the previous value. In some contemporary superscalar processors, the previous mapping is already read from the rename map table before updating it with the new mapping.

RF: When a predicted silent write issues, it indexes the physical register file using the previous mapping, to obtain the previous value of the logical register. If the previous value has not been produced yet, the direct check of the silent write is not performed. This approach is the most complex since it increases pressure on register read ports and may require changes to the select logic for read port arbitration.

ORT_dis: Slipstream's Operand Rename Table (ORT) is an existing component that detects dead writes and silent writes, an essential part of learning what to remove from the partial thread in the future [6]. It resembles an architectural register file, namely it is indexed by logical register and values are committed to it as instructions retire. These values facilitate detection of silent writes: a silent write is detected when its value matches the corresponding one in the ORT. The ORT can be leveraged for direct checks of silent writes, before or at retirement. In the case of ORT_dis, the ORT is queried at dispatch. When a predicted silent write dispatches, it reads the corresponding value from the ORT. The immediately previous write may not have retired yet in which case the ORT does not have the value we need to compare with. This can be determined by adding a mapping field to the ORT which indicates the committed mapping. If the previous mapping obtained at dispatch matches the ORT committed mapping, the ORT value is the immediately previous value we want to compare with. Otherwise the direct check of the silent write is not performed.

ORT_exe: This approach is the same as ORT_dis except the ORT is queried when the predicted silent write executes. Waiting until execution increases the chance that the previous value is in the ORT.

ORT_ret: The ORT is queried for the previous value when the predicted silent write retires. In fact, this query is already done by the IR-detector in the course of learning about past silent writes, but previously this query was not also exploited to perform direct checks of predicted silent writes (and possible recovery) before retirement. ORT_ret essentially comes for free in the baseline Slipstream implementation. While this direct check detects and recovers from faults in all correctly predicted silent writes, without a history buffer recovery

model, the direct check is too late to recover from faults originating in the backward slices of the silent writes. Fortunately, results in Section 2.6 show that good coverage is primarily needed for the silent writes themselves.

ORT_all: This is a combination of ORT_dis, ORT_exe, and ORT_ret: the ORT is queried at all stages until the previous value becomes available in the ORT. The previous value is guaranteed to be available by the time the predicted silent write retires (ORT_ret).

2.3.3.2 Method for Directly Checking Silent Stores

A predicted silent store is converted to a load in the load/store pipeline to obtain the previous value at the store address. If it is truly silent, there is no net increase in cache bandwidth since the converted load replaces the store.

2.4 Novel Coverage Analysis

Instead of literally injecting faults in the simulator, each and every instruction is viewed as potentially faulty. Accordingly, each instruction is said to be a "candidate faulty instruction". Each candidate faulty instruction is scrutinized to determine whether or not it is directly or indirectly checked before it retires. In Slipstream, duplicated instructions and some non-duplicated instructions (explained below) are directly checked via pairwise comparisons whereas all other non-duplicated instructions are indirectly checked, as discussed in Section 2.2. If the analysis framework determines that a candidate faulty instruction is checked either directly or indirectly before retirement, the instruction is included in coverage since recovery would succeed in this case (the faulty instruction would be prevented from retiring, or the faulty instruction is architecturally dead [10]).

Analysis is straightforward for duplicated instructions and directly-checked non-duplicated instructions. Directly-checked non-duplicated instructions include (1) confident branches and (2) confident silent writes and stores, only if direct checks of silent writes and stores are employed. These instructions are called “checkers” since they check themselves. Checkers are included in coverage if there are no coincident mispredictions (mispredictions cause losses in coverage, as discussed in Section 2.2.5).

The complexity of the analysis framework stems from non-duplicated instructions that are not directly checked. These instructions are said to be “non-checkers” since they cannot check themselves. The analysis technique examines a forward slice of each non-checker instruction in the full thread, sufficient for safely (conservatively) determining whether or not the instruction is checked by checker descendants before retirement or does not need to be checked (architecturally dead). Note that this is only a measurement technique in the simulator, not a hardware mechanism (although it could be a useful mechanism for explicitly deferring retirement of non-checker instructions for higher coverage).

Because of the possibility of masking consumers, using only the first completed checker descendant is not a safe indicator that its non-checker ancestor is checked before retirement. A masking consumer is one which cannot detect a faulty producer because it produces a correct output despite an incorrect input. While it may seem like the faulty producer does not matter, it depends on whether or not there are additional, non-masking consumers. Ideally, the analysis should identify the first completed non-masking checker descendant, since it truly checks its non-checker ancestor. However, determining whether or not an instruction is non-masking is sometimes difficult because it depends on specific values

and fault locations interacting with the operation type. Moreover, a conservative value-agnostic measure of coverage is preferred, as a bound.

Therefore, instead of explicitly searching for the first completed non-masking checker descendant, the analysis considers all direct/indirect checker descendants under the assumption that one or more are unknowingly masking. More formally, a finite forward slice is identified, whose leaf instructions consist of only checker instructions: (1) duplicated instructions, (2) confident branches, (3) confident dead writes, and/or (4) confident silent writes or stores, only if direct checks of these are employed.

An example forward slice of a non-checker instruction, A, is shown in Figure 2-10. The terminal instructions (leaf instructions) of the slice include three duplicated instructions D, F, and H. In addition, there are two other terminal instructions, a confident branch E and confident dead write G. There is a very simple criterion for knowing when a forward slice is fully formed. It is fully formed when there are no "live" non-checker instructions in the slice. If there are no live non-checker instructions, there are no possible places in the slice where additional terminal instructions (checker descendants) can be added. In the example, non-checker instructions A, B, and C have all been killed and hence the forward slice of A is fully formed (note that G must also be killed, confirming its checker status as a confident dead write). In addition, the forward slices of non-checker instructions B and C are fully formed, too.

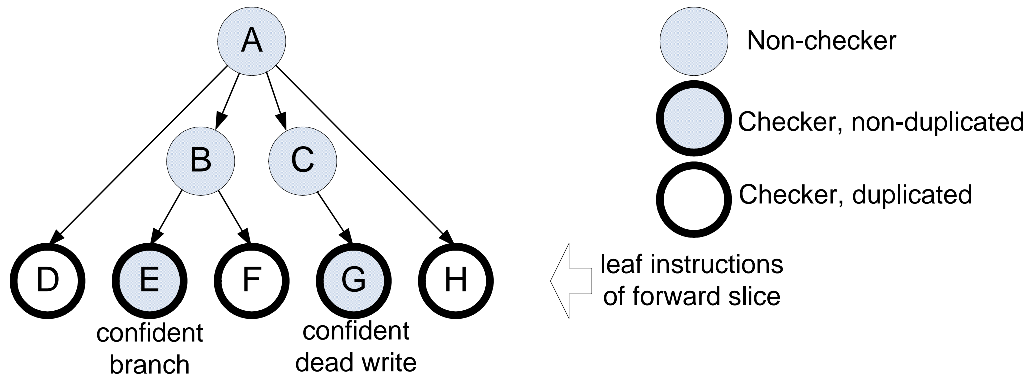


Figure 2-10. Forward slice example.

If the following three criteria are met before a non-checker instruction is retired, then the non-checker instruction is covered, assuming ROB-flush or History Buffer recovery:

Its forward slice is fully formed, i.e., there are no live non-checker instructions remaining in the ongoing slice, since these represent possible places to add other checkers.

All predictions in the slice are correct (mispredictions reduce coverage, as explained in Section 2.2.5, and this is accounted for in overall coverage).

All terminal instructions of the slice have completed execution, not only checking themselves but also indirectly checking all non-checker ancestors in the slice.

If none of the terminal instructions of a fully-formed misprediction-free slice detect a fault, it confirms that either (i) the non-checker ancestors are fault-free or (ii) the non-checker ancestors may be faulty but all of their checker descendants are masking instructions, so they can be safely retired despite being faulty.

2.5 Simulation Environment

A detailed cycle-level simulator is used that models three different execution modes on a common simultaneous multithreading (SMT) microarchitecture, shown in Table 2-1.

Single: Only a single thread.

Slip: Slipstream execution using a partial leading thread and a full trailing thread.

Slip-Full: Slipstream execution using a full leading thread and a full trailing thread (100% coverage). This models an optimized full duplication approach [23][20], since the trailing thread is "accelerated" by outcomes from the leading thread. Full duplication is achieved by turning off instruction removal from the leading thread.

The Slipstream implementation mirrors the microarchitecture described in previous work. The parameters for Slipstream components and A-stream/R-stream memory management are the same as in previous work [6][17]. SPEC2K integer benchmarks are used. They are compiled with the Simplescalar gcc compiler [3] for the PISA ISA. The compiler optimization level is –O3. Reference inputs are used. For the runs, 1 billion instructions are skipped and the following 100 million instructions are simulated. Only 10 of 12 integer benchmarks are run because eon and crafty do not compile.

Table 2-1. SMT microarchitecture for all three modes.

L1 I & D caches	64KB, 4-way, 64B line, LRU, L1hit = 1 cycle, L1miss/L2hit = 10 cycles
L2 unified cache	1MB, 8-way, 64B line, LRU, L1miss/L2miss = 100 cycles
branch predictor	gshare, 16-bit history, 2^{20} entries
superscalar core	reorder buffer (ROB): 256 load/store queue: 64 issue queue: 64 dispatch/issue/retire bandwidth: 8 cache ports: 4 read/write

2.6 Results

2.6.1 Instruction Breakdown

To understand coverages of different Slipstream variants in subsequent sections, it is useful to refer to the breakdown of retired instructions in Figure 2-11. Three types of checkers are shown at the bottom of each bar: (1) Dup – duplicated instructions, (2) B – confident branches, (3) D – confident dead writes. The next two types may be checkers or non-checkers, depending on whether or not direct checks are employed: (4) SS – confident silent stores, (5) SW – confident silent writes. The next four types are non-checkers: (6) bs_B – in backward slice of confident branch, (7) bs_SS – in backward slice of confident silent store, (8) bs_SW – in backward slice of confident silent write, (9) Other – this includes smaller components such as bs_D (backward slice of confident dead write) and instructions in backward slices of multiple types. This breakdown chart will be referred to when explaining coverages.

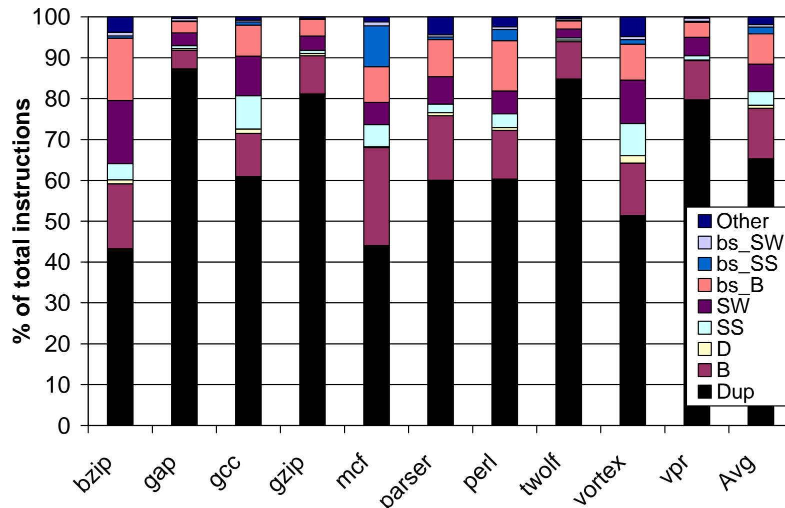


Figure 2-11. Breakdown of retired instructions.

2.6.2 Slipstream with Base Recovery

In this section coverages are presented for various Slipstream implementations that all use the base recovery model. The base recovery model restarts from the checker that detects the fault, whether or not the original faulty instruction is in the backward slice of the checker. Thus, in all cases, coverage is limited to only checker instructions (those that can check themselves): Dup, B, D, and possibly SS, SW.

Coverages are shown in Figure 2-12. The first bar (PriorWork) shows the coverage of baseline Slipstream as predicted by prior work [27][16]. The coverage only includes Dup, which is an underestimate. The second bar (Slip) shows the true coverage of baseline Slipstream, corresponding to all checker instructions (Dup, B, D). Coverage of Slip is 78% on average, whereas PriorWork puts coverage at only 65% on average.

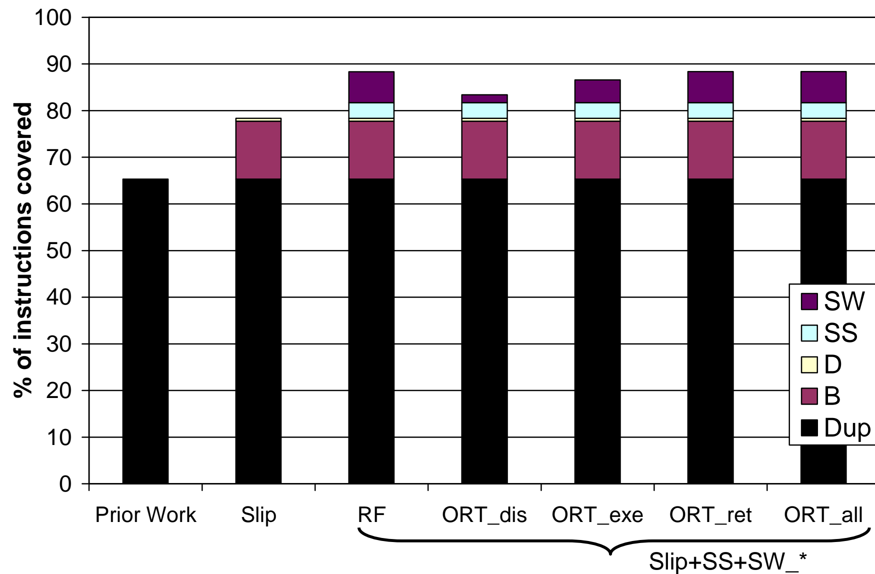


Figure 2-12. Coverages of Slipstream with base recovery.

The next five bars show Slipstream augmented with direct checks of both silent stores and silent writes (Slip+SS+SW_*). There are five bars corresponding to the five methods for direct checking of silent writes, using notation from Section 2.3.3.1.

With Slip+SS+SW_*, coverage includes a varying percentage of SW instructions depending on when the direct checks of SW instructions are performed. As expected, querying the ORT at dispatch (ORT_dis) yields the least coverage of SW instructions since the ORT often does not yet contain the previous value for comparison.

Querying the ORT at retirement (ORT_ret) yields total coverage of SW instructions since the previous value is always available for comparison at that time. The downside of ORT_ret is that, unless history buffer recovery is used, no bs_SW instructions are covered despite total coverage of SW instructions. This downside is not apparent in this section due to the inadequate base recovery model (no backward slice instructions are covered at all). Referring to Figure 2-13, notice that bs_SW constitutes less than 1% of instructions whereas SW itself constitutes 7% of instructions, on average. It can be concluded that any increase in bs_SW coverage that may be afforded by earlier direct checks of SW (e.g., ORT_dis or ORT_exe) is not worth the greater loss in SW coverage. While RF achieves the same coverage of SW instructions as ORT_ret, RF is more complex to implement. Summing up, ORT_ret is the best choice in that it yields the highest incremental coverage and it is very cheap to implement in an existing Slipstream implementation.

Slip+SS+SW_ORT_ret yields the highest coverage with base recovery: 88%.

2.6.3 Slipstream with New Recovery

This section explores the coverage benefits of the new Slipstream recovery approaches. All coverages in Figure 2-13 are for Slip+SS+SW_ORT_ret, varying only the recovery model. The first bar reiterates coverage of base recovery (base). The next three bars show coverages with ROB-head (RH) recovery, i.e., when a checker instruction detects a fault, the processor restarts both threads from the instruction at the head of the ROB. The number after RH indicates whether or not our second technique, ROB occupancy management, is used and what the occupancy threshold is. RH0 means there is no threshold hence no occupancy management. RH32 and RH48 correspond to ROB-head recovery with ROB occupancy targets of 32 and 48 instructions by the full thread, respectively.

RH0 increases coverage to 95%, up from 88% with base recovery. This is due to partial coverage of non-checkers, for the first time. RH32 and RH48 increase coverage of non-checkers even further. By deferring retirement of non-checkers, RH32/RH48 increases the chances that non-checkers are indirectly checked before retirement. Coverage reaches 97% and 98% for RH32 and RH48, respectively. However, a potential drawback of ROB occupancy management is performance degradation, which is explored further in Section 2.6.4.

The history buffer (HB) approach provides a performance-friendly alternative to ROB occupancy management. The final two bars in Figure 2-13 show history buffers of 16 instructions (HB16) and 32 instructions (HB32). Coverage is 98% and 99% for HB16 and HB32, respectively.

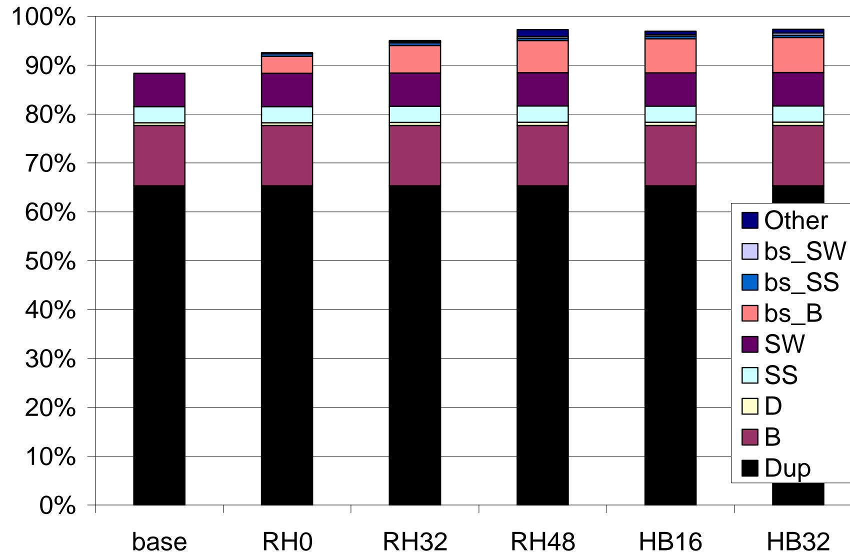


Figure 2-13. Coverage of Slip+SS+SW_ORT_ret for various recovery approaches..

2.6.4 Performance

Figure 2-14 shows the performance of (1) single-thread execution (Single), (2) Slipstream with full duplication (Slip-Full), (3) Slipstream with partial duplication and base recovery (Slip:base), and (4) Slipstream with partial duplication, using Slip+SS+SW_ORT_ret and various recovery models (Slip+:*). Single and Slip-Full show the two extremes between full performance and full fault tolerance. Slip-Full achieves 100% fault tolerance, but at the price of 14% average slowdown. Among benchmarks with IPCs above 2.5 (bzip, gap, gcc, perl, and vortex), where the processor is utilized moderately well by a single thread, the average slowdown of Slip-Full is 18.5%.

The first three bars (Single, Slip-Full, Slip:base) are used to categorize benchmarks into three groups. Mcf, twolf, and vpr show little difference in performance among different threading scenarios. These benchmarks utilize the processor poorly with a single thread, hence adding another thread has little impact. Gap is in its own category. Gap utilizes the

processor quite well with a single thread and both Slip-Full and Slip:base cause a significant slowdown. Slip:base underperforms because gap has little instruction removal as can be seen in Figure 2-11: 87% of instructions are duplicated. Finally, bzip, gcc, gzip, parser, perl, and vortex show mild to significant slowdown with Slip-Full, yet Slip:base performs comparably to Single. In some cases Slip:base outperforms Single, which is not unexpected since Slipstream is known to enhance performance on both CMP and SMT substrates [16].

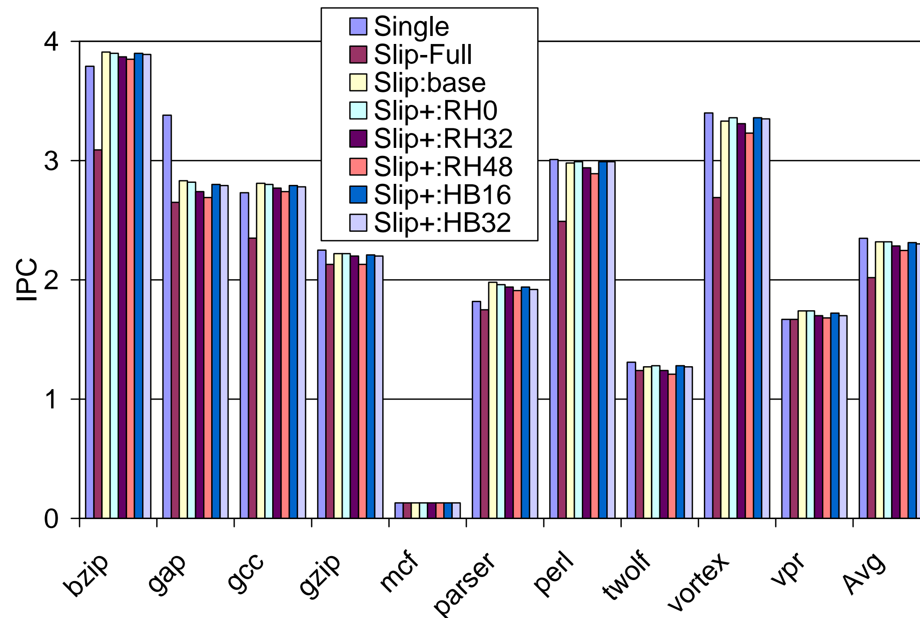


Figure 2-14. Performance comparisons.

On average, Slip:base is only 1.3% slower than Single, whereas Slip-Full is 14% slower. Among 2.5+ IPC benchmarks, Slip:base is only 2.7% slower, whereas Slip-Full is 18.5% slower.

All benchmarks show the expected behavior for Slip+. First, there is a gradual decrease in performance from base through RH0, RH32, RH48. RH0 flushes more instructions than base, and RH32/RH48 defer retirement. Second, the history buffer approaches HB16 and HB32 climb back to the performance of base because retirement is not

deferred. Perl, twolf, and vortex show slightly more performance with Slip+:* compared to Slip:base. This is due to slight timing perturbations caused by converting silent stores to loads in Slip+.

2.7 Using Only Branch Predictions

In this section, the forward-slice analysis framework is applied to estimate the dynamic instruction coverage of a ReStore-like architecture. In particular, there is only a single thread and confident branch predictions are used to check corresponding branches and their backward slices.

Figure 2-15 shows the coverage and breakdown of covered instructions. All correctly predicted branches are covered because a correct prediction will differ from a wrong outcome. Thus, both confident (B) and unconfident (b) correctly predicted branches are included in coverage. However, only mispredictions among confidently predicted branches cause a rollback to an earlier point in the program. Therefore, the forward-slice analysis only considers confident correctly predicted branches (B) as checkers, i.e., terminals of a forward slice. Accordingly, an arbitrary instruction is not considered checked until its forward slice has only confident correctly predicted branches (B) and/or dead writes (D) for leaves. In Figure 2-15, arbitrary instructions that are successfully checked in this way constitute the bs_B category (in backward slice of B). The number of bs_B instructions increases with larger rollback distances, as shown for various recovery models (RH0, HB32, HB64, HB128). On average, Single+B (single thread with B as checkers) yields 33% coverage with HB128 recovery (D, B, b, and bs_B are covered).

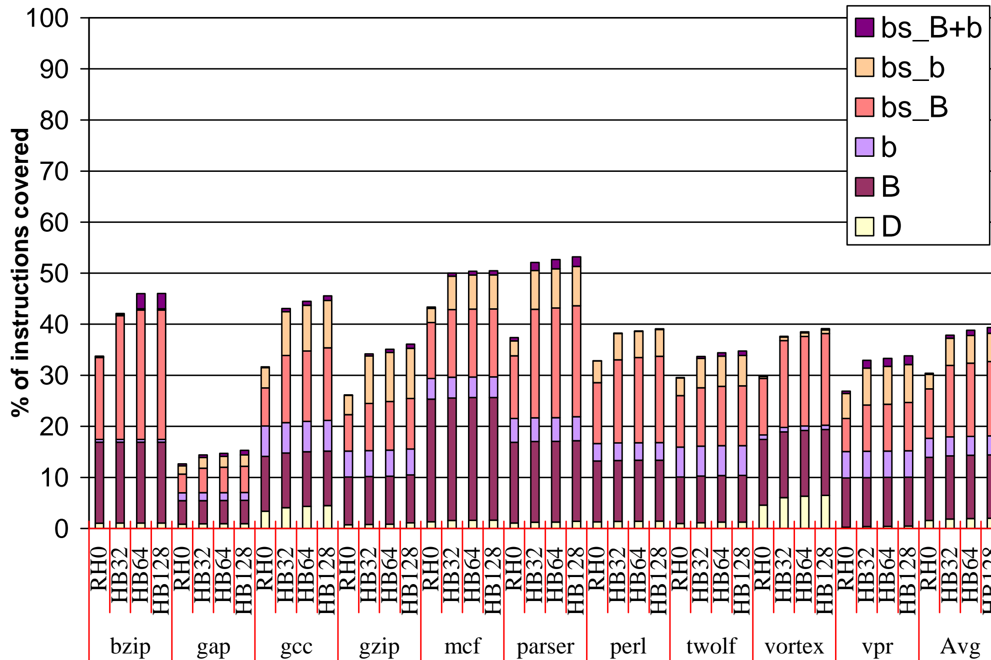


Figure 2-15. Coverage for Single+B(+b).

Coverage can be maximized by rolling back for all branch mispredictions. This means both confident (B) and unconfident (b) correctly predicted branches are checkers (that is, all correctly predicted branches). This covers additional instructions, bs_b (in backward slice of b) and bs_B+b (in backward slice of B and b) in Figure 2-15. On average, coverage for Single+B+b is 40% with HB128 recovery.

A more optimistic analysis assumes no fault masking by B (or b) instructions, meaning that an arbitrary instruction is considered checked by the first B (or b) instruction encountered in its forward slice. Corresponding optimistic coverages are shown in Figure 2-16 (There is no bs_B+b category since the first B or b instruction is used to check). On average, optimistic coverages for Single+B and Single+B+b are 50% and 60%, respectively, with HB128 recovery.

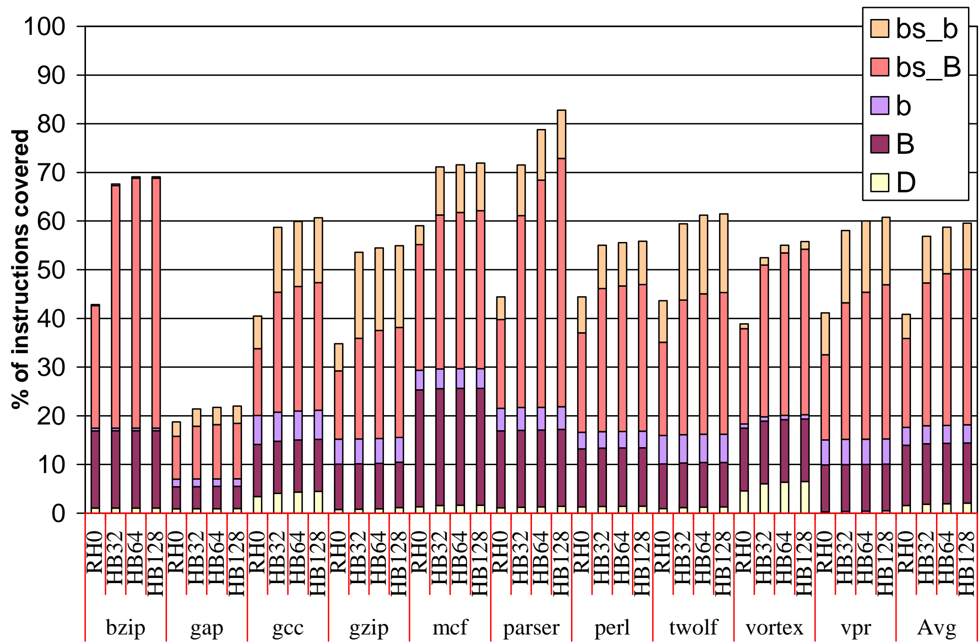


Figure 2-16. Optimistic coverage for Single+B(+b).

Performance of Single+B and Single+B+b is shown in Figure 2-17. The performance of Single+B+b is not shown with a dedicated bar, rather, it is shown with a negative error bar with respect to the Single+B bar. Peformance of Single (no coverage) and Slip-Full (full coverage) are shown for comparison.

The performance degradation of Single+B, with respect to Single, is mild. Rollbacks are rare because only mispredictions among confidently predicted branches cause rollbacks. As expected, the performance degradation increases slightly with more distant rollbacks (from RH0 to HB128).

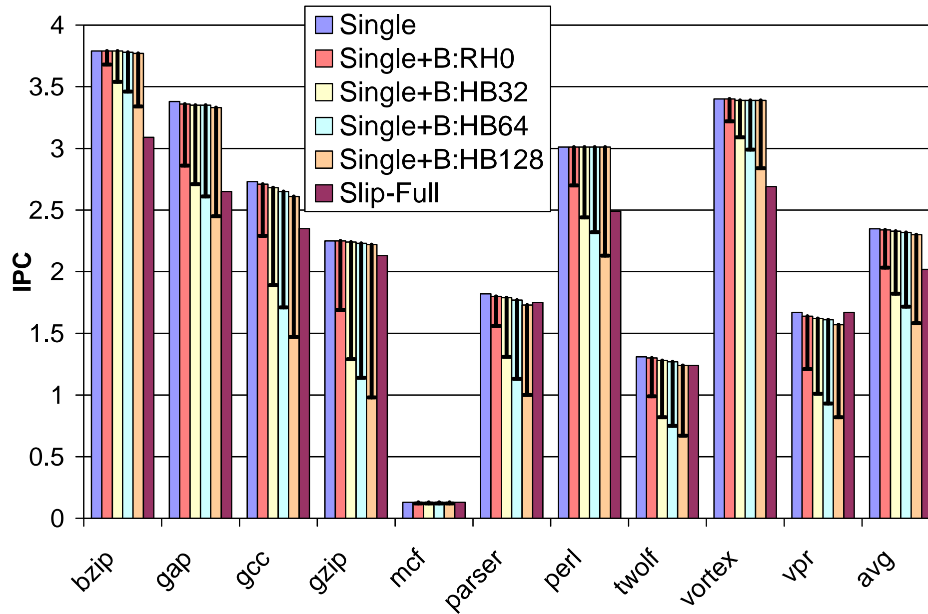


Figure 2-17. Performance of Single+B(+b).

The performance degradation of Single+B+b (negative error bar) is severe, because all mispredictions cause rollbacks. For the same reason, performance is very sensitive to the rollback distance. Nonetheless, for some benchmarks and rollback distances, Single+B+b outperforms Slip-Full (albeit with less coverage).

2.8 Summary

Prediction-based checking is a promising new direction in efficient fault tolerance. In this chapter, it was shown for the first time that the combination of confident predictions and partial duplication can approximate the fault tolerance of full duplication. Slipstream is a convenient substrate for testing this hypothesis, as it embodies the notion of comparing a full thread with an independent, reduced, predictive thread. A thorough dissection of the four prediction scenarios in Slipstream revealed near-100% fault detection capability despite duplicating as few as 43% of instructions. Slipstream's recovery and checking implementation was then revamped with a suite of strategies that made it possible to nearly

fully capitalize on the excellent fault detection capability. Exploiting prediction-based PRT on Slipstream yielded instruction coverages of 99% with performance close to a single thread.

Chapter 3: Inherent Time Redundancy

In this chapter, a new approach is proposed that exploits repetition inherent in programs to provide low-overhead transient fault protection in a processor. Programs possess inherent time redundancy (ITR): the same instructions are executed repeatedly at short intervals. This program repetition presents an opportunity to discover low-overhead fault checks in a processor. The key idea is to record microarchitectural events which depend purely on program instructions, and confirm the occurrence of those events when instructions repeat. The recorded microarchitectural events must depend purely on instructions being executed. For example, the decode signals generated upon fetching and decoding an instruction are the same across all instances. Recording and confirming them to be the same can detect faults in the fetch and decode units of a processor. Similarly, recording and confirming correct issue ordering among instructions in a trace can detect faults in the out-of-order scheduler of a processor, similar to Timestamp-based Assertion Checking (TAC) [43].

In this chapter, it will be shown that ITR can be leveraged to detect transient faults in the fetch and decode units of a processor pipeline, avoiding costly approaches like structural duplication or explicit time redundant execution.

The chief contributions made in this chapter are as follows:

1. A new fault tolerance approach is proposed based on inherent time redundancy (ITR) in programs. The key idea is to record and confirm microarchitectural events that depend purely on program instructions.

2. An ITR cache is proposed to record microarchitectural events pertaining to a trace of instructions. The key novelty is that misses in the ITR cache do not directly lead to a loss in fault detection. Only evictions of unreferenced, missed instances lead to a loss in fault detection coverage. Microarchitectural support is developed to use the ITR cache for protecting the fetch and decode units of a high-performance processor.
3. On fault detection, it is shown that accurate identification of the correct recovery strategy is possible: either a lightweight flush and restart of the processor, or a more expensive program restart.
4. It is shown that the ITR-based approach compares favorably to conventional approaches like structural duplication and time redundant execution, in terms of area and power.

The rest of the chapter is organized as follows. Section 3.1 characterizes repetition in SPEC2K benchmarks. Section 3.2 discussed the ITR cache. Section 3.3 discusses detailed microarchitectural support to exploit ITR for protecting the fetch and decode units of a superscalar processor. In Section 3.4 the ITR cache design space is explored to achieve high fault coverage. In Section 3.5, fault injection experiments are performed to further evaluate fault coverage. In Section 3.6, area and power overheads of the ITR approach are compared to other fault tolerance approaches. Section 3.7 summarizes the chapter.

3.1 Characterizing Program Repetition in SPEC2K Benchmarks

Repetition in SPEC2K programs is characterized in Figure 3-1 (integer benchmarks) and Figure 3-2 (floating point benchmarks). Instructions are grouped into traces that terminate either on a branching instruction or on reaching a limit of 16 instructions. The

graphs plot the number of dynamic instructions contributed by static traces. Static instructions are unique instructions in the program binary, whereas dynamic instructions correspond to the instruction stream that unfolds during execution of the program binary.

A relatively small number of static instructions contribute a large number of dynamic instructions. For instance, in most integer benchmarks, less than five hundred static traces contribute nearly all dynamic instructions (e.g., in bzip, 100 static traces contribute 99% of all dynamic instructions). Gcc and vortex are the only exceptions due to the large number of static traces. Floating point benchmarks are even more repetitive, as seen in Figure 3-2 (e.g., in wupwise, 50 static traces contribute 99% of all dynamic instructions).

An important aspect of repetition is the distance at which traces repeat. This is characterized in Figure 3-3 (integer benchmarks) and Figure 3-4 (floating point benchmarks). Here, instructions are grouped into traces like before, and the number of dynamic instructions between repeating traces is measured. The graphs show the number of dynamic instructions contributed by all static traces that repeat within a particular distance. Distances are shown at increasing intervals of five hundred dynamic instructions.

As seen, there is a high degree of ITR in programs. In all integer benchmarks, except perl and vortex, 85% of all dynamic instructions are contributed by traces repeating within five thousand instructions, four of them reaching that target within one thousand instructions.

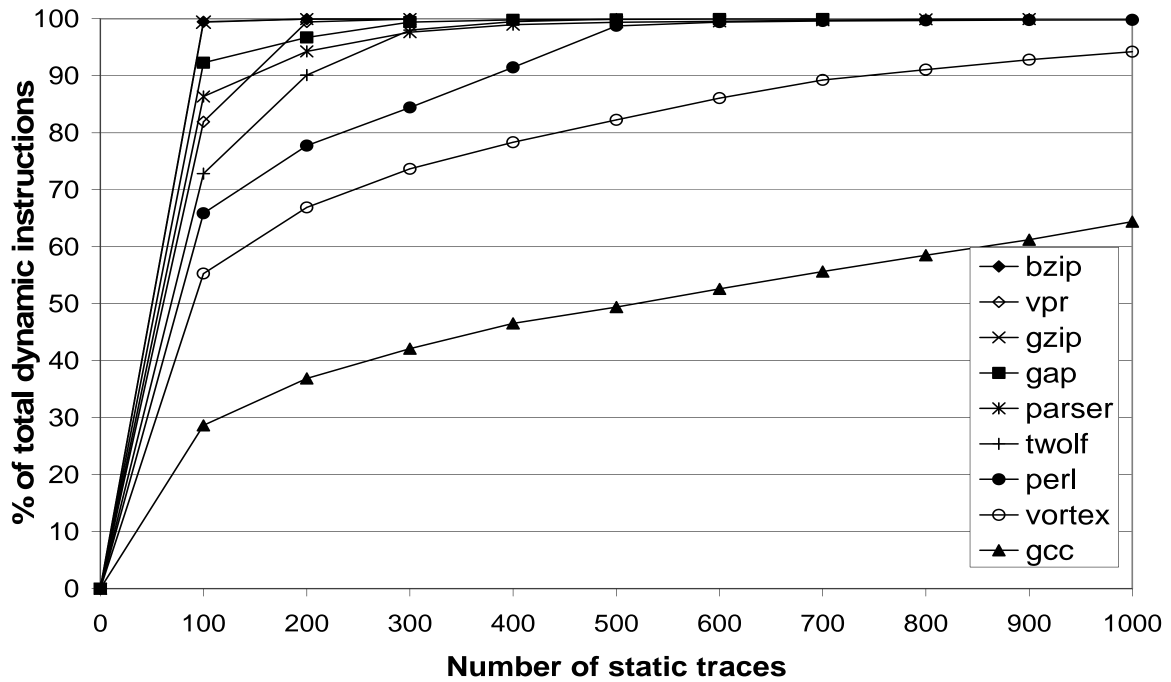


Figure 3-1. Dynamic instructions per 100 static traces (integer benchmarks).

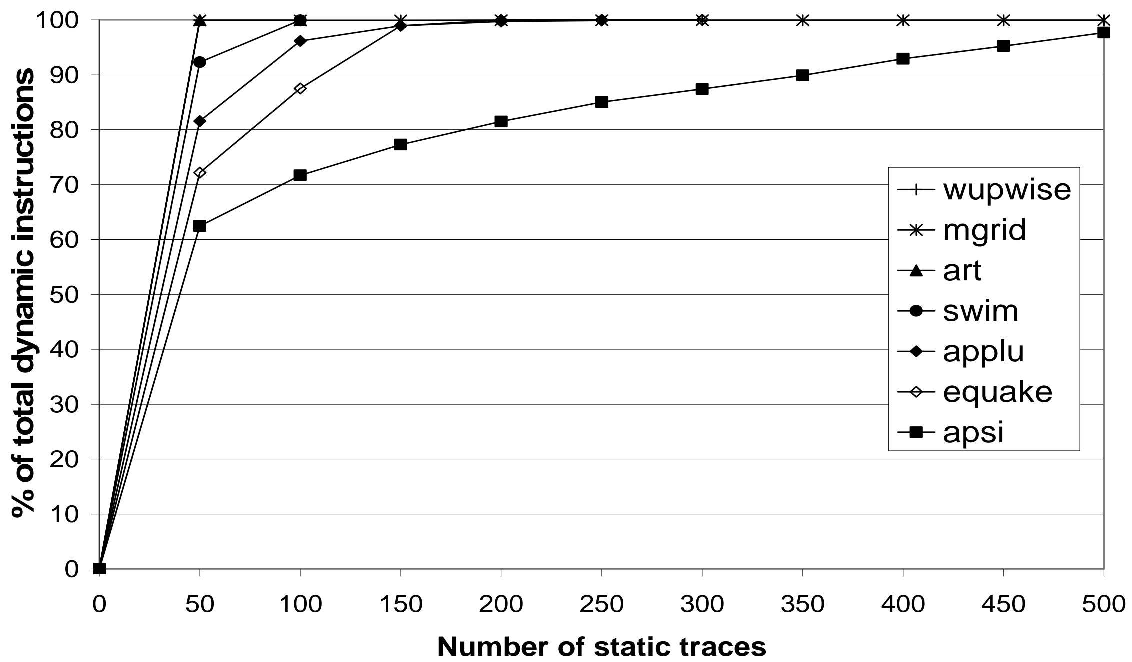


Figure 3-2. Dynamic instructions per 50 static traces (floating point benchmarks).

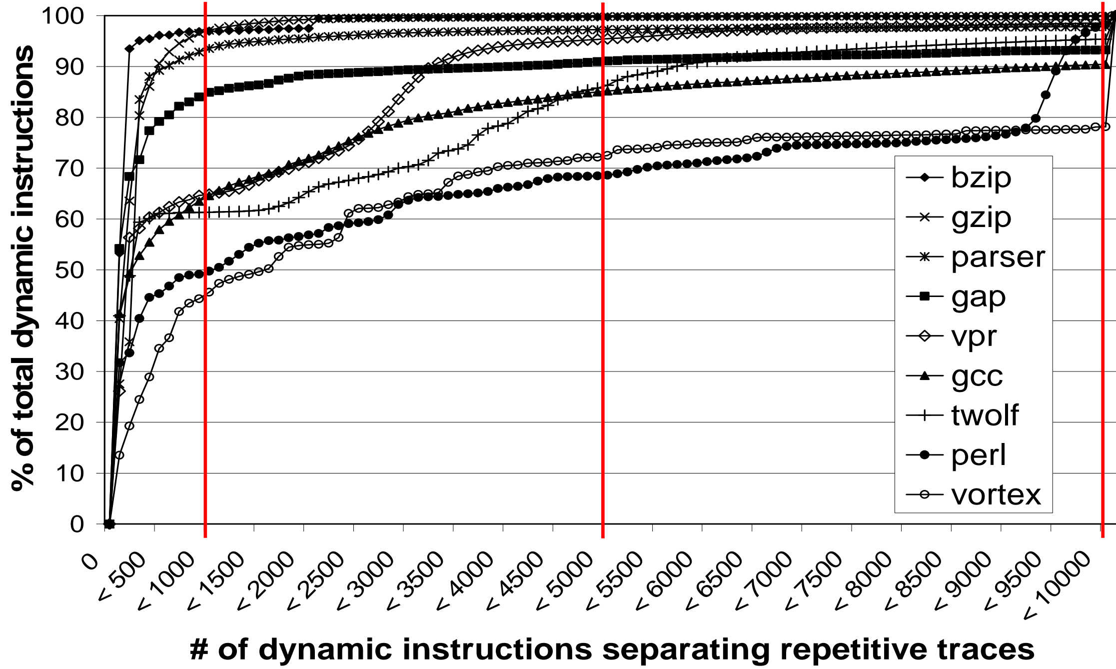


Figure 3-3. Distance between trace repetitions (integer benchmarks).

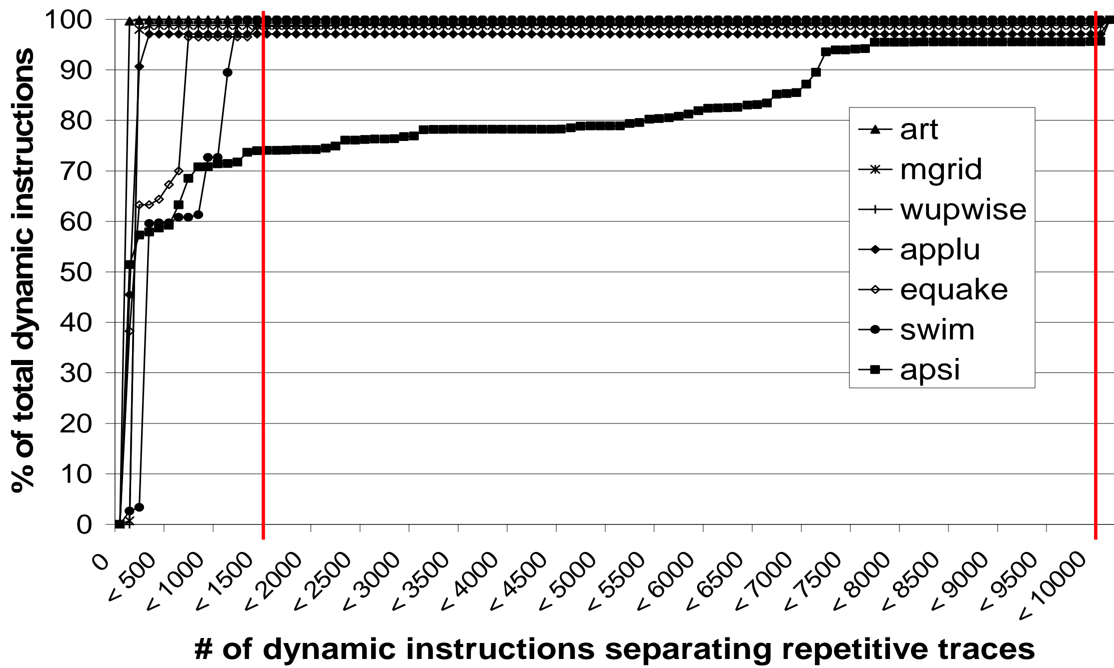


Figure 3-4. Distance between trace repetitions (floating point benchmarks).

In all floating point benchmarks, except apsi, nearly all dynamic instructions are contributed by repetitive traces with high proximity (within 1500 instructions).

3.2 ITR Cache

The main idea for leveraging ITR for fault tolerance is to record and confirm microarchitecture events that occur while executing highly repetitive instruction traces. The fact that relatively few static traces contribute heavily to the total instruction count, suggests that a small structure is sufficient to record events for most benchmarks. I propose to use a small cache to record microarchitecture events during repetitive traces. The cache is indexed with the starting program counter (PC) of a trace. A miss in the cache indicates the unavailability of a counterpart to check the correctness of the microarchitectural events. However, misses do not always lead to loss of fault detection. A future hit to a trace that previously missed in the cache can detect anomalies during execution of both the missed instance and the newly executed instance of the trace. In a single-event upset model, a reasonable assumption for fault studies, the two instances will differ if there is a fault. However, if a missed instance is evicted from the cache before it is accessed, it constitutes a loss in fault detection, since a fault during the missed instance goes undetected. Based on this, even benchmarks with a large number of static traces and mild proximity (e.g., gcc) can get reasonable fault detection coverage with small event caches.

3.3 Microarchitecture Support for ITR

In this section and following subsections, microarchitecture support is added to use ITR to extend transient fault protection to the fetch and decode units of a superscalar processor. Signals generated by the decode unit for instructions in a trace are combined to

generate a signature. The signature is stored in a small cache, called the ITR cache. On the next occurrence of the trace, the signature is re-generated and compared to the signature stored in the ITR cache. A mismatch indicates a transient fault either in the fetch or the decode unit of the processor. On fault detection, safe recovery may be possible by flushing and restarting the processor from the faulting trace, or the program must be aborted through a machine check exception. Insight is provided into diagnosing a fault and defines criteria to accurately identify fault scenarios where safe recovery is possible, and where aborting the program is the only option.

The architecture of a superscalar processor, augmented with support for exploiting ITR, is shown in Figure 3-5. The shaded components are newly added to protect the fetch and decode units of the processor using ITR. The new components are described in subsections 3.3.1 through 3.3.5.

3.3.1 ITR Signature Generation

As seen in Figure 3-5, signals from the decode unit are redirected for signature generation. The signals are continuously combined until the end of each trace. The end of a trace is signaled upon encountering a branching instruction or the last of 16 instructions. On a trace ending instruction, the current signature is dispatched into the ITR ROB. The signature is then reset and a new start PC is latched in preparation for the next trace.

Signature generation could be done in many ways. I chose to simply bitwise XOR the signals of a new instruction with corresponding signals of previous instructions in the trace. For a given trace, if a fault on an instruction in the fetch unit or the decode unit causes a wrong signal to be produced by the decode unit, then the signature of the trace would differ

from that of a fault-free signature. Even multiple faulty signals in a trace would lead to a difference in signature, unless an even number of instructions in the trace produce a fault in the same signal. Using XOR to produce the signature loses information about the exact instruction that caused a fault. But this precision is not required as long as recovery is cognizant that a fault could be anywhere in the trace and rollback is prior to the trace. For a single-event upset model, this overall approach is sufficient for detecting faults on an instruction of a trace in the fetch and decodes units.

3.3.2 ITR ROB and ITR cache

Trace signatures are dispatched into the ITR ROB, when trace termination is signaled. The ITR ROB is sized to match the number of branches that could exist in the processor, since every branch causes a new trace.

Since a trace is terminated on a branch, its ITR ROB entry is noted in the branch's checkpoint to facilitate rollback to the correct ITR ROB entry on branch mispredictions.

Each ITR ROB entry stores the start PC and the signature of a trace. An ITR ROB entry also contains control bits (chk, miss, retry), which indicate the status of checking the trace with the copy in the ITR cache.

The ITR cache stores signatures of previously encountered traces and is indexed with the start PC of a trace. Each trace in the ITR ROB accesses the ITR cache at dispatch. This ensures that reading the ITR cache is complete before the instructions in the trace are ready to commit. If the trace hits, the signature is read from the ITR cache and checked with the signature of the trace. Regardless of the outcome, the chk (for checked) bit is set in the

corresponding ITR ROB entry. If it's a mismatch, the retry bit of the ITR ROB entry is set. If the trace misses, the miss bit of the ITR ROB entry is set.

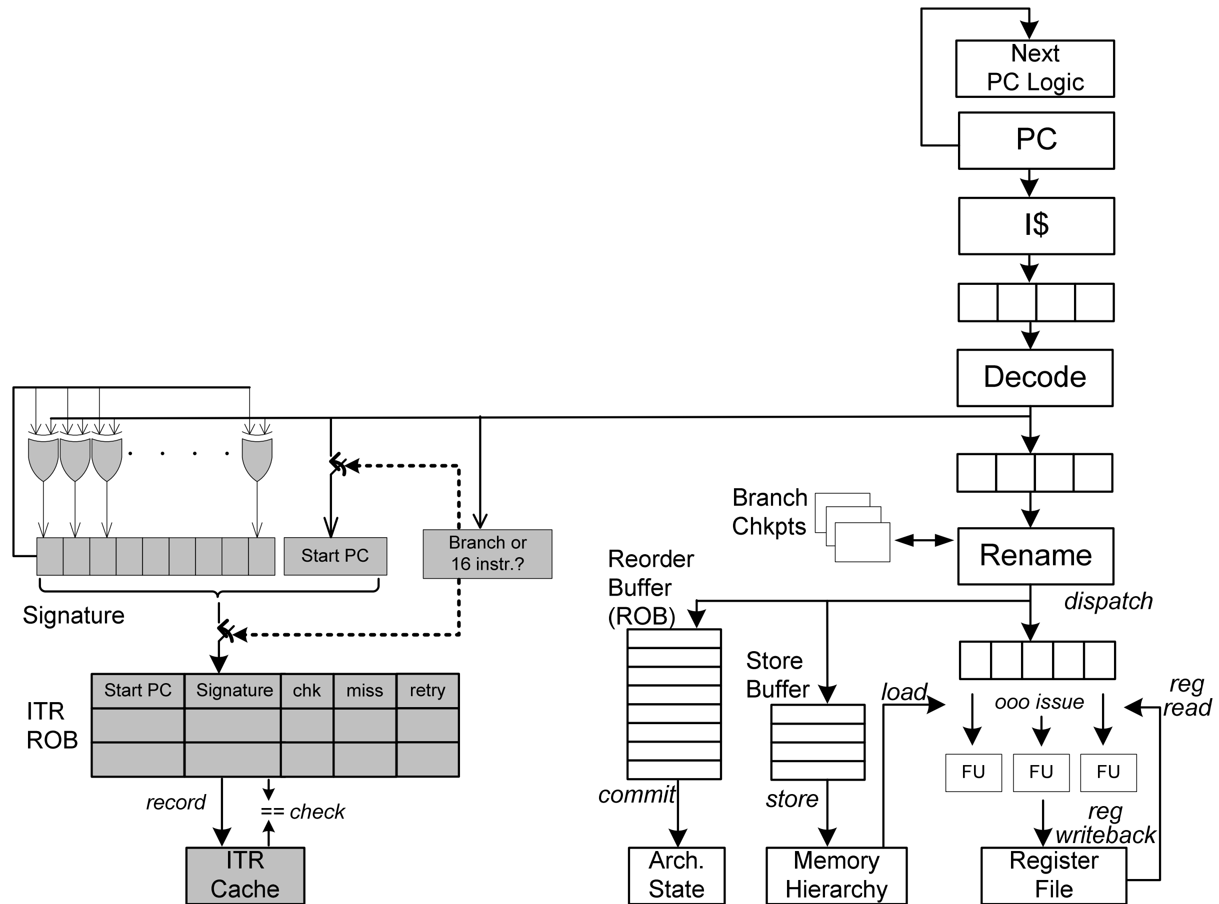


Figure 3-5. Superscalar processor augmented with ITR support.

The ITR ROB enables the commit logic of the processor to determine whether the trace of the currently committing instruction has been formed, whether it is has been checked, whether it is faulty, etc. The only extra work for the commit logic is to poll the head entry of the ITR ROB when an instruction is ready to commit. It polls to see if the miss bit or the chk bit of the ITR ROB head entry is set. If neither is set, commit is stalled until one of the bits is set. If the miss bit is set, then a write to the ITR cache is initiated and commit from the main ROB progresses normally. If the chk bit is set, and additionally the retry bit is not

set, then instructions are committed from the main ROB normally. If the retry bit is set, it indicates a transient fault occurred in either the new trace or the previous trace that stored its signature in the ITR cache. To confirm which trace instance is faulty, the processor is flushed and restarted from the start PC of the new trace. If the signatures mismatch again, then it is clear the previous trace executed with a fault. Since this means the processor's architectural state could be corrupted, a machine check exception is raised and the program is aborted. However, if the signatures match after the retry, it means the new trace was faulty, and recovery through flushing and restarting the processor was successful. In all cases, when a trace-terminating instruction is committed from the main ROB, the ITR ROB head entry is freed.

3.3.3 Fault Detection and Recovery Coverage

Writing to the ITR cache involves replacing an existing, least recently used (LRU) trace signature. Evicting an existing trace signature has implications on the fault detection coverage, i.e., the number of instructions in which a fault can be detected. If a trace's signature is not referenced before being evicted, it amounts to a loss in fault detection coverage. To prevent this, a bit could be added to each cache line to indicate that it is checked and the replacement policy could be modified to evict the LRU trace that has been checked. This optimization is not studied and instead the loss in fault detection coverage for different cache configurations is reported. Moreover, this policy is not applicable to direct mapped caches and breaks down when no ways of a set are checked yet.

ITR cache misses decrease the fault recovery coverage, i.e., the number of instructions in which a fault can be detected and successfully recovered by flushing and restarting the

processor. This is because on a miss, an unchecked trace signature is entered into the cache. If the unchecked trace is faulty, the fault is only detected in the future by the next instance of the trace. However, since the faulty trace has already corrupted the architectural state, the program has to be aborted. In Section 3.4, the fault coverage for different ITR cache configurations is measured.

Recovery coverage can be enhanced through a coarse-grained checkpointing scheme (e.g., [54][55]). The key idea is to take a coarse-grain checkpoint when there are no unchecked lines in the ITR cache. The number of unchecked lines could be tracked. Once it reaches zero, a coarse-grain checkpoint could be taken. Then in cases where the lightweight processor flush and restart is not possible, recovery can be done by rolling back to the previously taken coarse-grain checkpoint instead of aborting the program.

3.3.4 Faults on ITR Components

The new ITR components do not make the processor more vulnerable to faults, assuming a single-event upset model. A fault on signature generation components will be detected as a signature mismatch. A fault on the latched start PC is not a concern. If its signature matches the faulty start PC's signature, the fault gets masked. If it mismatches, the fault is detected. If it misses in the ITR cache, the next instance of the faulty PC will either detect it or mask it. The control bits chk, miss and retry can be protected using one-hot encoding. The possible states are: {none set – 0001, chk and retry set – 0010, chk set and retry not set—0100, miss set – 1000}. Faults on the ITR cache will cause false machine check exceptions when they are detected, i.e., a retry will indicate a fault on the trace signature in the ITR cache and a machine check exception will be raised, as described in

Section 3.3.2. This can be avoided by parity-protecting each line in the ITR cache. On a signature mismatch, retry is attempted. If the signature mismatches again, then parity is checked on the trace signature in the cache. A parity error indicates an error in the ITR cache and not the previous instance of the trace. Successful recovery involves invalidating the erroneous line in the cache, or updating it with the signature of the new trace.

3.3.5 Faults on the Program Counter (PC)

A fault on the PC or the next-PC logic causes incorrect instructions to be fetched from the I-cache.

If the disruption is in the middle of a trace, then its signature will be a combination of signals from correct and incorrect instructions, and will differ from the trace's fault-free signature. In this case, a PC fault is detected by the ITR cache.

If the disruption is at a natural trace boundary, then a wrong trace is fetched from the I-cache. Since the signature of the wrong trace itself is unaffected by the fault, it will agree with the ITR cache. Hence, the PC that starts a trace at a natural trace boundary represents a vulnerability of the ITR cache, and needs other means of protection. For natural trace boundaries caused by branches, substantial protection of the PC already exists, because the execution unit checks branch targets predicted by the fetch unit. For natural trace boundaries caused by the maximum trace length, protection of the PC is possible by adding a simple retirement PC and asserting that a committing instruction's PC matches the retirement PC. The retirement PC is updated as follows. Sequential committing instructions add their length (which can be recorded at decode for variable-length ISAs) to the retirement PC and branches update the retirement PC with their calculated PC. Comparing a committing

instruction's PC with the retirement PC will detect a discontinuity between two otherwise sequential traces.

3.4 The ITR Cache Design Space

As noted in Section 3.3.3, evictions of unreferenced lines from the ITR cache cause a loss in fault detection coverage, and misses in the ITR cache cause a loss in fault recovery coverage. In this section, different ITR cache configurations are tried and the loss in fault detection coverage and fault recovery coverage are measured for each design point. Loss in coverage is measured by noting the number of instructions in vulnerable traces.

For experiments, SPEC2K integer and floating point benchmarks are used that are compiled with the Simplescalar gcc compiler for the PISA ISA [3]. The compiler optimization level is –O3. Reference inputs are used. In the runs, 900 million instructions are skipped and 200 million instructions simulated.

Two ITR cache parameters are varied, (1) Associativity: direct mapped (dm), 2-way, 4-way, 8-way, 16-way and fully associative (fa), and (2) Cache size: 256, 512 and 1024 signatures. Figure 3-6 shows the loss in fault detection coverage and Figure 3-7 shows the loss in fault recovery coverage for the various cache configurations. For a given associativity, a smaller cache increases the number of evictions of unreferenced ITR signatures and the number of ITR cache misses. The corresponding increase in coverage loss is shown stacked for the various cache sizes.

Bzip, gzip, art, mgrid and wupwise have negligible coverage loss for all ITR cache configurations. For clarity, they are not included in the graphs. Their excellent ITR cache behavior can be explained by referring back to Figure 3-3 and Figure 3-4, which characterize

ITR in benchmarks. In these benchmarks, traces repeat in close proximity and such traces contribute to nearly all the dynamic instructions.

In fact, coverage loss for all benchmarks correlates with their characteristics in Figure 3-3 and Figure 3-4. In perl and vortex, traces that repeat far apart contribute to a large number of dynamic instructions. Correspondingly, they have the highest loss in fault coverage. Cache capacity has a big impact on mitigating this loss. For example, in vortex, for a direct-mapped cache, increasing the cache capacity to 1024 signatures from 256 signatures decreases the loss in fault detection coverage to 12% from 33%.

Gcc, twolf and apsi also have a notable number of traces that repeat far apart, and experience a loss in fault coverage. They also benefit significantly from increasing the cache capacity. Table 3-1 is referred for insight. It shows the total number of static traces for all benchmarks. Notice for vortex and perl, the number of static traces (2,655 and 1,704) is higher than the capacity of all the ITR caches simulated. Their poor trace proximity exposes this capacity problem. Far-apart repeating traces get evicted before they are accessed again, leading to a notable loss in fault coverage. Increasing the cache capacity somewhat makes up for the poor proximity and, hence, has a big impact on reducing coverage loss. Gcc confirms our hypothesis that proximity amongst traces is a strong factor. Even though it has far more traces than vortex and perl (24,017), it has lower coverage loss for a given cache configuration as a result of its better trace proximity. Mgrid is another example. It has negligible coverage loss for all ITR cache configurations even though it has a relatively high number of static traces (798). Again, proximity amongst its traces is excellent. The remaining

benchmarks have a small loss in fault coverage which can be overcome with bigger caches or higher associativity.

Note that the loss in fault coverage should not be interpreted as a conventional cache miss rate, i.e., it does not correspond to signatures that missed on accessing the ITR cache. Firstly, the loss in fault detection coverage (Figure 3-6) corresponds to signatures that were evicted from the ITR cache before being referenced. Secondly, both the loss in fault detection coverage and the loss in fault recovery coverage are influenced by the number of instructions in signatures, which is not uniform across all signatures. These factors may explain why, in some benchmarks, higher associativity sometimes happens to show slightly higher loss in fault coverage than lower associativity.

Table 3-1. Number of static traces for SPEC.

SPECInt	#static
bzip	283
gap	696
gcc	24017
gzip	291
parser	865
perl	1704
twolf	481
vortex	2655
vpr	292

SPECfp	#static
applu	282
apsi	1274
art	98
equake	336
mgrid	798
swim	73
wupwise	18

An important point is that the loss in fault detection coverage is significantly lesser than the loss in fault recovery coverage for all benchmarks. This is because all ITR cache misses lead to a loss in recovery coverage, but only those missed traces that are then evicted before being referenced lead to a loss in detection coverage.

Across all benchmarks, for a 2-way associative cache with 1024 signatures, the average loss in fault detection coverage is 1.3% with a maximum loss of 8.2% for vortex. The corresponding numbers for loss in fault recovery coverage are 2.5% average and 15% maximum for vortex.

In general, programs with less repetition or greater distance between repeated traces would have a higher loss in fault coverage. One possible solution to mitigate this is to redundantly fetch and decode traces only on a miss in the ITR cache, still achieving the benefits of ITR but falling back on conventional time redundancy when inherent time redundancy fails. After the signature of the re-fetched trace is checked against the ITR cache, instructions in that trace are discarded from the pipeline. Another possible solution is to have a fully duplicated frontend, like in the IBM S/390 G5 processor [51], but use the ITR cache to guide when the space redundancy should be exercised (for significant power savings). The use of ITR as a filter for selectively exercising time redundancy or space redundancy is an interesting direction for future research.

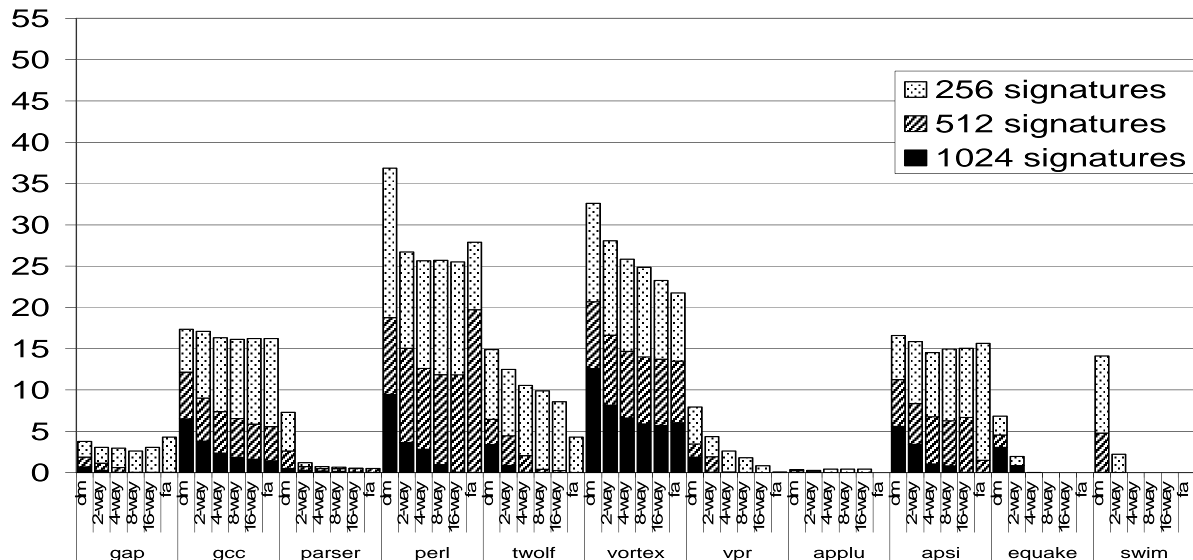


Figure 3-6. Loss in fault detection coverage.

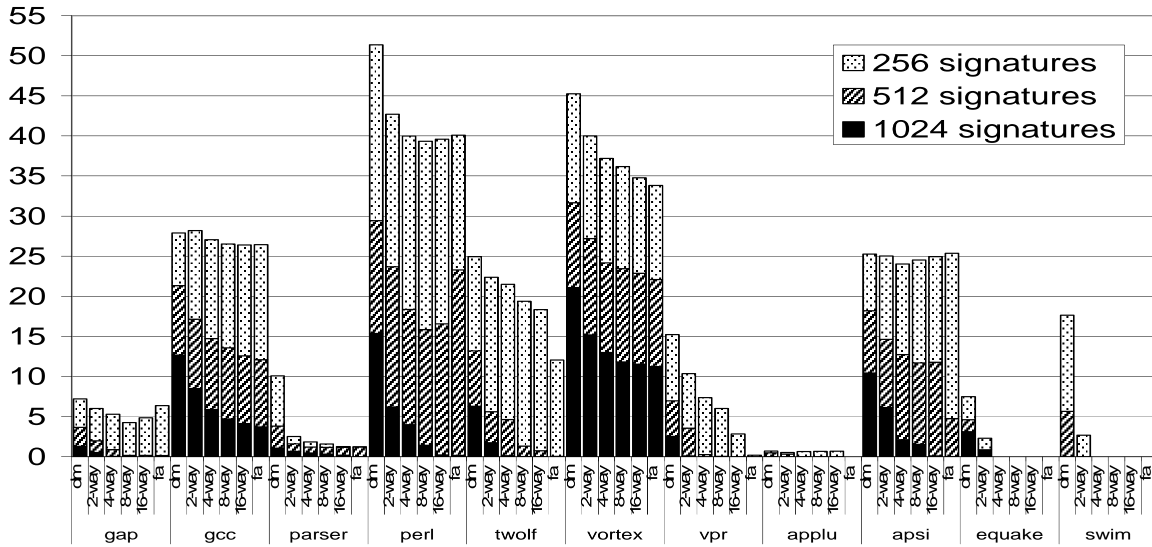


Figure 3-7. Loss in fault recovery coverage.

3.5 Fault Injection Experiments

Fault injection is performed on a detailed cycle-level simulator that models a microarchitecture similar to the MIPS R10000 processor [41]. The modeled microarchitecture configuration is shown in Table 3-2.

Table 3-2. Microarchitecture configuration

L1 I & D Caches	64KB, 4-way, 64B line, LRU, L1hit = 1 cycle, L1miss/L2hit = 10 cycles
L2 Unified Cache	1MB, 8-way, 64B line, LRU, L1miss/L2miss = 100 cycles
Branch Predictor	gshare, 16-bit history, 2^{20} entries
Superscalar Core	reorder buffer (ROB): 64 dispatch/issue/retire bandwidth: 4 per cycle

A subset of SPEC2K Integer and FP benchmarks are used for evaluation. For each benchmark, one thousand faults are randomly injected on the decode signals from Table 3-3. Injecting a fault involves flipping a randomly selected bit. A separate “golden” (fault-free) simulator is run in parallel with the faulty simulator. When an instruction is committed to the

architectural state in the faulty simulator, it is compared with its golden counterpart to determine whether or not the architectural state is being corrupted. Any fault that leads to corruption of architectural state is classified as a potential silent data corruption (SDC) fault. Likewise, if no corruption of architectural state is observed for a set period of time after a fault is injected (the observation window), it is classified as a masked fault. In this study, an observation window of one million cycles is used.

An injected fault may lead to one of six possible outcomes, depending on (1) whether the fault is detected by an ITR check ("ITR") or undetected within the scope of the observation window ("MayITR") or undetected for sure ("Undet"), and (2) whether the fault corrupts architectural state ("SDC") or not ("Mask"). Based on this, the six possible outcomes are ITR+SDC, ITR+Mask, MayITR+SDC, MayITR+Mask, Undet+SDC, and Undet+Mask.

ITR+SDC outcomes are further qualified with the possibility of recovery (ITR+SDC+R) or only detection (ITR+SDC+D). On detecting a fault through ITR, if the signature accessing the ITR cache is faulty as opposed to the signature within the cache, then, the fault is recoverable by flushing the ROB (discussed in Section 3.3.3).

Two more fault checks are added to support the experiments. A watchdog timer check (wdog) is added to detect deadlocks caused by some faults (e.g., faulty source registers). A sequential-PC check (spc) is added at retirement (discussed in Section 3.3.5) to detect faults pertaining to control flow.

Table 3-3. List of decode signals.

Field	Description	Width
opcode	instruction opcode	8
flags	decoded control flags (is_int, is_fp, is_signed/unsigned, is_branch, is_uncond, is_ld, is_st, mem_left/right, is_RR, is_disp, is_direct, is_trap)	12
shamt	shift amount	5
rsrc1	source register operand	5
rsrc2	source register operand	5
rdst	destination register operand	5
lat	execution latency	2
imm	immediate	16
num_rsrc	number of source operands	2
num_rdst	number of destination operands	1
mem_size	size of memory word	3
	Total width	64

As a final note about methodology, all faults are injected on correct-path instructions. This way, the chance that a fault surfaces as an error is increased, thus testing the fault detection capabilities of ITR better.

In the following experiments, a two-way set-associative ITR cache is used for holding 1024 signatures. The breakdown of fault injection outcomes is shown in Figure 3-8. Fault injection results are shown for the same set of SPEC benchmarks whose coverage results are reported in Section 3.4. As seen, a large percentage of injected faults are detected through the ITR cache (95.4% on average). On average, 32% of the injected faults are detected and recovered by ITR that would have otherwise led to a SDC (ITR+SDC+R). Only a small percentage (1% on average) of SDC faults detected through ITR is not recoverable (ITR+SDC+D). A large percentage of faults that are detected by ITR happen to get masked

(59.4% on average). When a fault is injected on a decode signal that is not relevant to the instruction being decoded or does not lead to an error (e.g., increasing lat, the execution latency, only delays wakeup of dependent instructions), then the fault gets masked, but the signature is faulty and gets detected by the ITR cache. A noticeable fraction of faults (3% on average) are detected and recovered by ITR that would have otherwise led to a deadlock (ITR+wdog+R), highlighting another important benefit.

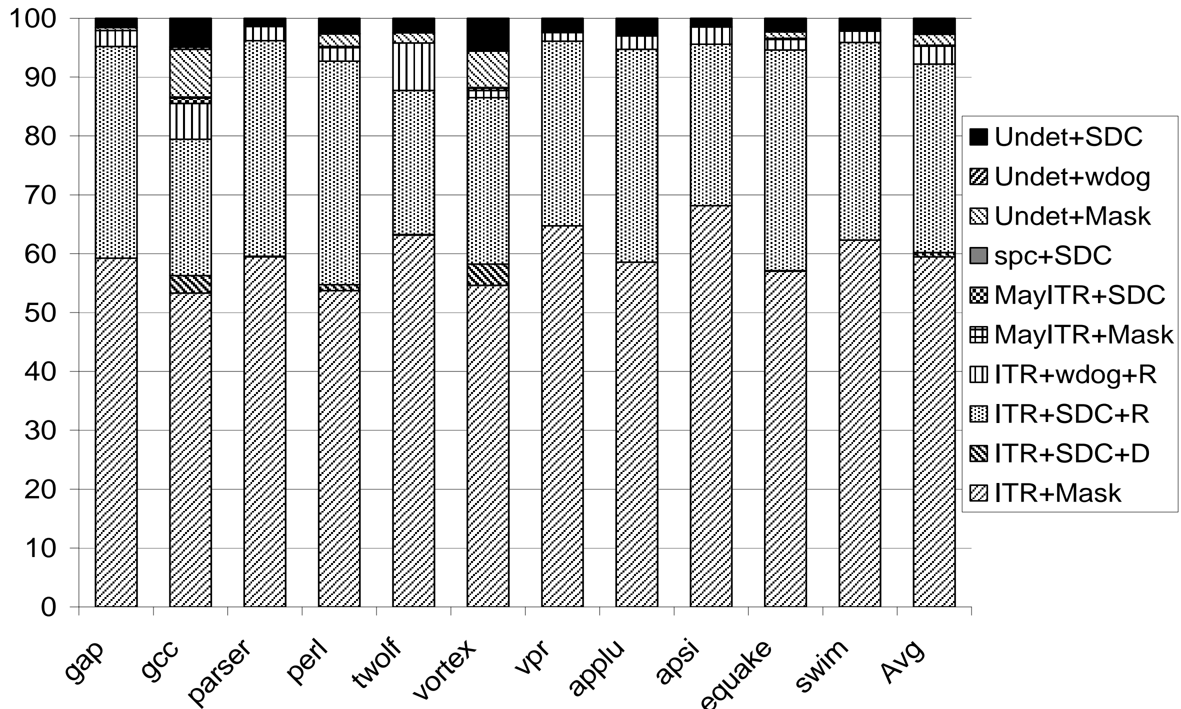


Figure 3-8. Fault injection results.

The fraction of faults undetected by ITR within the observation window (MayITR+*) is negligible. This indicates that a one million cycle observation window is sufficient.

Interestingly, the sequential PC check detected a small fraction of faults (0.1% on average) that ITR alone could not detect (spc+SDC). The sequential-PC check mainly detected faults on the is_branch control flag, which indicates whether or not an instruction is

a conditional branch. Consider the following fault scenario. Suppose that the fetch unit predicts an instruction to be a conditional branch (BTB hit signals a conditional branch and gshare predicts taken). Suppose the instruction is truly a conditional branch (BTB correct) and is actually not taken (gshare incorrect). Then suppose that a fault causes is_branch to be false instead of true. First, this fault causes a SDC because the branch misprediction will not be repaired. Second, because is_branch is false, the retirement PC is updated in a sequential way. The spc check will fire in this case, because the next retiring instruction is not sequential. Note that if the prediction was correct (actually taken), the spc check still fires, but this is a masked rather than SDC fault.

On average, 4.5% of injected faults go undetected by ITR. Only about 2.6% of the faults lead to SDC and are not detected by ITR (Undet+SDC). A very small fraction of faults (0.1% on average) lead to a deadlock that is not detected by ITR but is caught by the watchdog timer. The remaining undetected faults are masked (on average, 1.8% of all faults).

3.6 Area and Power Comparisons

Structural duplication can be used to protect the fetch and decode units of the processor. In the IBM S/390 G5 processor [41], the I-unit, comprised of the fetch and decode units, is duplicated and signals from the two units are compared to detect transient faults. However, this direct approach has significant area and power overheads. It is attempted to compare the area and power overhead of the ITR cache with that of the I-unit, to see whether or not the ITR-based approach is attractive compared to straightforward duplication. The die photo of the IBM S/390 G5 shown in Figure 3-9 provides the area of the I-unit [41]. To estimate the area of the ITR cache, a structure is selected from the die photo that is similar in

configuration to the ITR cache. The branch target buffer (BTB) of the G5 has a configuration similar to the ITR cache: 2048 entries, 2-way associative, 35 bits per entry [52]. Based on the decode signals in Table 2, the size of the ITR signature is 64 bits. Though each ITR entry is almost twice as wide as the G5's BTB entry, only half as many entries as the BTB (1024 entries) are needed for good coverage, from results in Section 3.4 and Section 3.5.

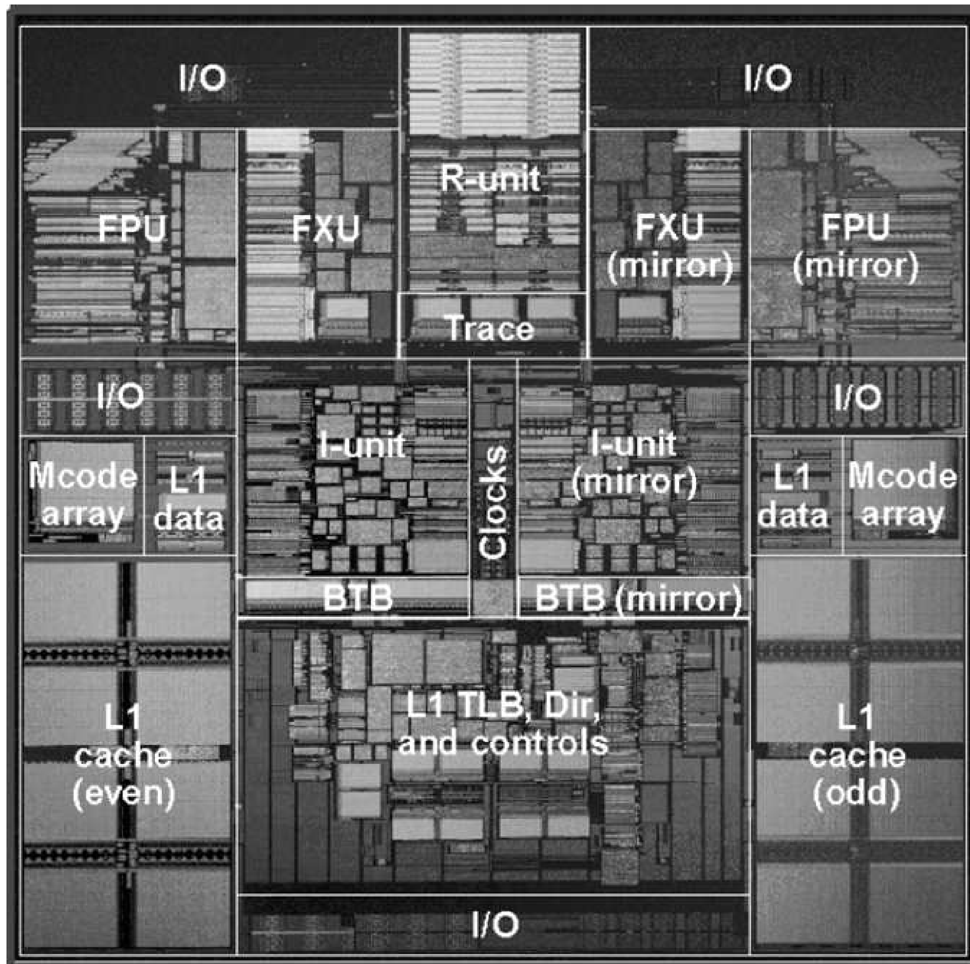


Figure 3-9. Die photo of the IBM S/390 G5 processor (Source: [51]).

The area of the I-unit from the die photo is 1.5 cm x 1.4 cm, i.e., 2.1 cm2. The area of the ITR-cache like BTB structure from the die photo is 1.5 cm x 0.2 cm, i.e., 0.3 cm2. The

ITR cache is about one seventh the area of the I-unit. Hence, the ITR-based approach to protect the frontend is more area-effective than structural duplication of the entire I-unit.

Next, the power-effectiveness of the ITR approach is evaluated. A major power overhead of structural duplication and conventional time redundancy is that of fetching an instruction twice from the instruction cache. Power consumption is modeled by measuring the number of accesses to the ITR cache and the instruction cache of the processor. Both cache models are fed into CACTI [56] to obtain the energy consumption per access. Multiplying the number of accesses with the energy consumed per access gives the energy consumption.

Due to lack of information on the instruction cache configuration of the IBM S/390 G5, the instruction cache of the IBM Power4 [53] is chosen. The configuration of the Power4 I-cache is: 64KB, direct-mapped, 128 byte line and one read/write port. The configuration of the ITR cache is: 8KB (1024 entries), 2-way associative, 8 byte line, and one read/write port (or one read and one write port). The 0.18 micron technology used in the IBM Power4 is chosen.

The CACTI numbers were: 0.87 nJ per access for the I-cache, 0.58 nJ per access (or 0.84 nJ for separate read and write ports) for the ITR cache. Overall energy consumption is shown in Figure 9. As seen, the ITR-based approach is far more energy efficient than fetching twice from the instruction cache. Note that the energy savings will be even greater if also considering the redundant decoding of instructions in the frontend in the case of structural duplication or traditional time redundancy.

As seen, the ITR cache is more cost-effective than straightforward space redundancy in the IBM mainframe processor [51]. However, it should be noted that complete structural duplication provides more robust fault tolerance than the ITR cache. They are two different design points in the cost/coverage spectrum.

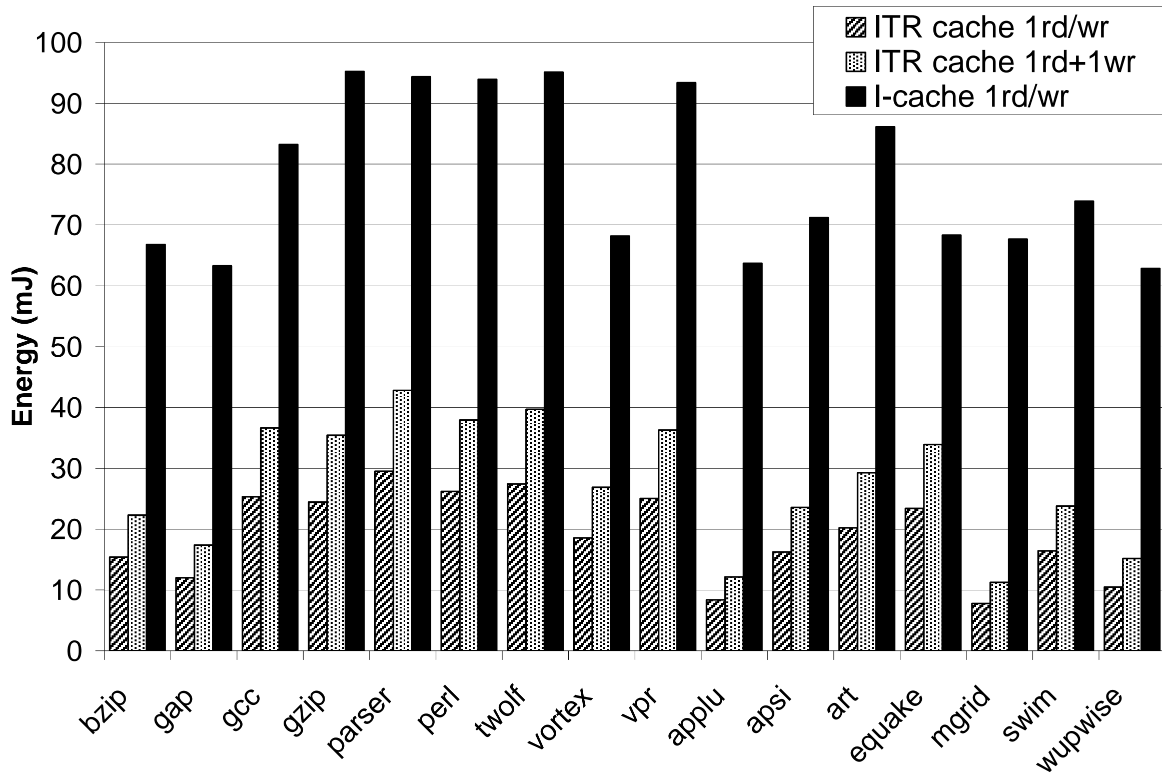


Figure 3-10. Energy of ITR cache vs. I-cache.

3.7 Summary

A new approach was introduced to develop low-overhead fault checks for a processor, based on inherent time redundancy (ITR) in programs. The ITR cache was proposed to store microarchitectural events that depend only upon program instructions. Its effectiveness was demonstrated by developing microarchitectural support to protect the fetch and decode units of the processor. Insights were given on diagnosing a fault to determine the correct recovery procedure. Fault detection coverage and fault recovery coverage obtained for a given ITR

cache configuration was quantified. Finally, it was shown that using the ITR-based approach is more favorable than costly structural duplication and traditional time redundancy.

Chapter 4: Register Name Authentication

Register Name Authentication (RNA) [43] aims to detect faults in register renaming logic and various renaming structures used in a contemporary superscalar processor [41], i.e., rename map table, architecture map table, shadow map tables (branch checkpoints), active list, and free list, that ultimately manifest as a fault in the destination physical register mapping of an instruction. A fault in the destination physical register of an instruction can lead to many anomalies depending on the location of the fault in the processor pipeline, and could ultimately corrupt the final program outcome. A list of possible faulty scenarios is listed below:

1. If the physical register tag carried by an instruction is faulty, the instruction may write its result to a wrong location, possibly clobbering a physical register being used by another instruction or depriving that instruction's consumers of the correct value.
2. If the physical register mapping in the rename map table (or shadow map tables) is faulty, the instruction that was mapped to the affected destination register appears to write to the wrong location. Hence, consumers of that instruction read values from the wrong location, ultimately producing wrong results.
3. If a physical register belonging to the free list of unmapped physical registers is faulty, an instruction that gets mapped to that faulty physical register may clobber another

instruction's physical register, or the faulty instruction's result might get clobbered by another instruction using the same (and correct)

In this chapter, low-overhead assertion checks are presented that can be introduced into the processor pipeline to detect such faults. RNA includes two assertion checks that authenticate an instruction's destination register mapping: the RNA previous mapping check and the RNA writeback state check. The fault tolerance capabilities of both the fault checks will be discussed and evaluated by performing fault injection on various renaming structures.

This chapter is organized as follows. RNA previous mapping check is discussed in Section 4.1 and the RNA writeback state check is discussed in Section 4.2. Section 4.3 discusses faults that only affect the source register mappings of instructions. Section 4.4 describes the fault injection experiments to evaluate RNA and discusses the results. Section 4.5 summarizes the chapter.

4.1 RNA Previous Mapping Check

When an instruction's logical destination register is renamed to a physical register, the rename map table is updated with the new register mapping. However, before updating the mapping, the previous mapping can be recorded in the instruction's entry in the active list. Thus, the instruction's entry in the active list has both the current and previous mappings of its logical destination register. Some superscalar processors already record the previous mapping in the active list, to facilitate freeing the previous mapping at retirement [41]. At retirement, before committing the instruction's current mapping to the architecture map table, the instruction's previous mapping can be confirmed to be the same as the corresponding mapping in the architecture map table.

This first RNA check can detect many faults that affect mappings in the rename map table, architecture map table, shadow map tables, and active list.

1. A fault that changes a mapping in the rename map table will be detected by the next instruction to update the mapping. The previous mapping recorded with this instruction (incorrect) will differ from the corresponding mapping in the architecture map table (correct). Detecting this fault is valuable for a couple of reasons, specifically, the fault may cause the wrong mapping to be freed and may cause consumers to receive a wrong source mapping.

2. A fault that changes a mapping in the architecture map table will be detected by the next instruction to update the mapping at retirement. This instruction's previous mapping (correct) will not match the corresponding mapping in the architecture map table (incorrect). If there is an exception before the instruction commits, the fault will not be detected and may have consequences for precise state. Otherwise the fault is detected and is not distinguishable from a fault in the rename map table.

3. A fault that changes a mapping in a shadow map survives if the shadow map is copied to the rename map table during a branch misprediction recovery. This scenario is then similar to a fault in the rename map table.

4. A fault that changes the previous mapping of an instruction in the active list will be detected when the instruction reaches retirement. Its previous mapping (incorrect) will not match the corresponding mapping in the architecture map table (correct). Detecting this fault is valuable because it causes the wrong mapping to be freed.

5. A fault that changes the current mapping of an instruction X in the active list will be detected by the next instruction Y with the same logical destination register. X will commit

the wrong mapping to the architecture map table. When Y commits, its previous mapping (correct) will not match the corresponding mapping in the architecture map table (incorrect). This scenario is basically the same as a fault in the architecture map table and is also vulnerable to the intervening exception case.

Summing up, there is inherent redundancy among the rename structures, and the first RNA check exploits this fact to detect inconsistencies in redundant destination register mappings that point to underlying faults.

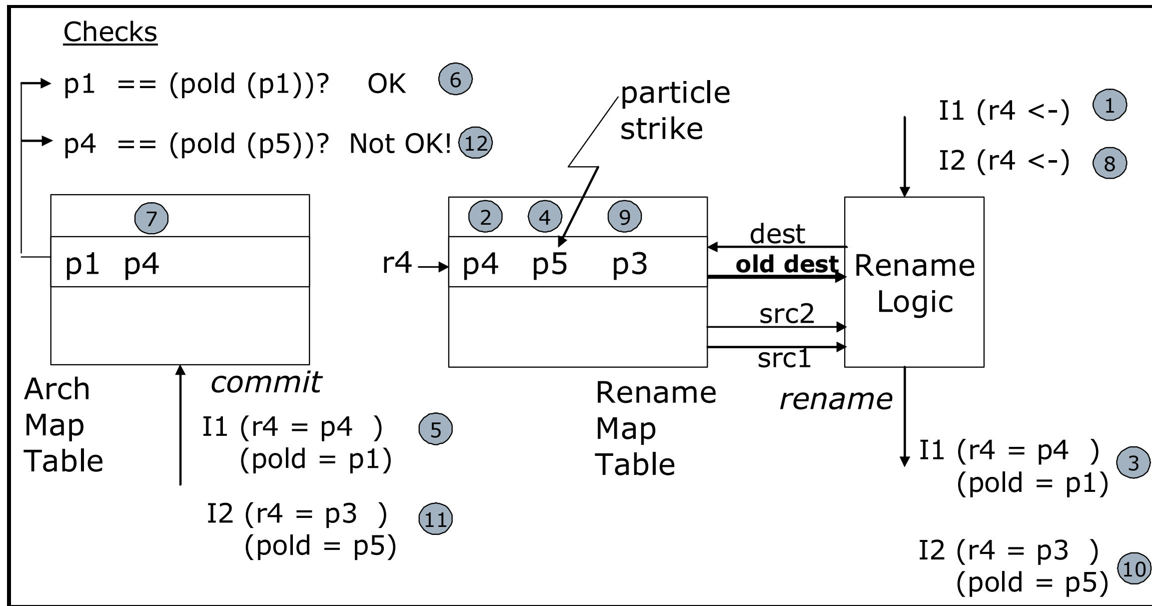


Figure 4-1. Fault detection using RNA's "previous mapping check".

The first fault scenario (a fault changes a mapping in the rename map table) is illustrated in Figure 4-1. Various events are indicated by numbers, indicating the order of events in time. Events 1 through 7 are for instruction I1 and events 8 through 12 are for instruction I2. At time 4, a fault causes the mapping r4 – p4 to change to r4 – p5, in the rename map table. When a later instruction I2 with logical destination register r4 is renamed, it records p5 as the old mapping of r4, instead of p4. At retirement of I2, the RNA check fails

to match I2's previous mapping of r4 (p5) to the mapping of r4 in the architecture map table (p4), hence detecting the fault. Again, catching this fault is valuable because the fault in the rename map table may have caused consumers of I1 to obtain a wrong source mapping (p5 instead of p4) and also causes I2 to free p5 instead of p4.

4.2 RNA Writeback State Check

RNA's previous mapping check can detect faults that cause inconsistencies between the architecture map table and other structures. However, it cannot detect faults in the free list or the destination renaming logic itself, i.e., the logic that presents a new mapping to the rename map table. An erroneous mapping this early is not distinguishable from a correct mapping. The erroneous mapping will be consistent among all the other structures (rename map, architecture map, shadow maps, and active list).

The RNA writeback state check detects such faults using the insight that they cause conflicts between the erroneous physical register mapping and the original physical register. Basically, an erroneously mapped physical register might be in use by another active instruction, or committed to architectural state, or still available in the free list. Confirming that a physical register is not ready and is also unavailable in the free list, at the register writeback stage of the processor pipeline, exposes the conflict, hence, detecting the fault.

Some superscalar designs already associate ready and free bits with each physical register [41], which can be leveraged to detect conflicts, hence, faults of the nature described above. The ready bit of a physical register is set when an instruction writes (or is about to write) to the physical register and is cleared when the physical register is added back to the

free list. The free bit of a physical register is set when it is added back to the free list and is cleared when it is popped from the free list.

The RNA writeback state check confirms neither of the bits is set before writing to a physical register. The intuition behind the RNA writeback state check is as follows. Suppose that the destination renaming logic produces a faulty physical register tag. There are two possibilities regarding the faulty tag , (1) it is in the free list, or (2) it is already being used by another instruction or is part of the architectural state. A set free bit at writeback detects the case of writing to a free register and a set ready bit detects the case of writing to a physical register that is already being used or part of the architectural state. For the case where a register is assigned to two instructions, the assertion check will fire when either the correct instruction or the faulty instruction writes back, depending on who writes last (the last writer observes a set ready bit).

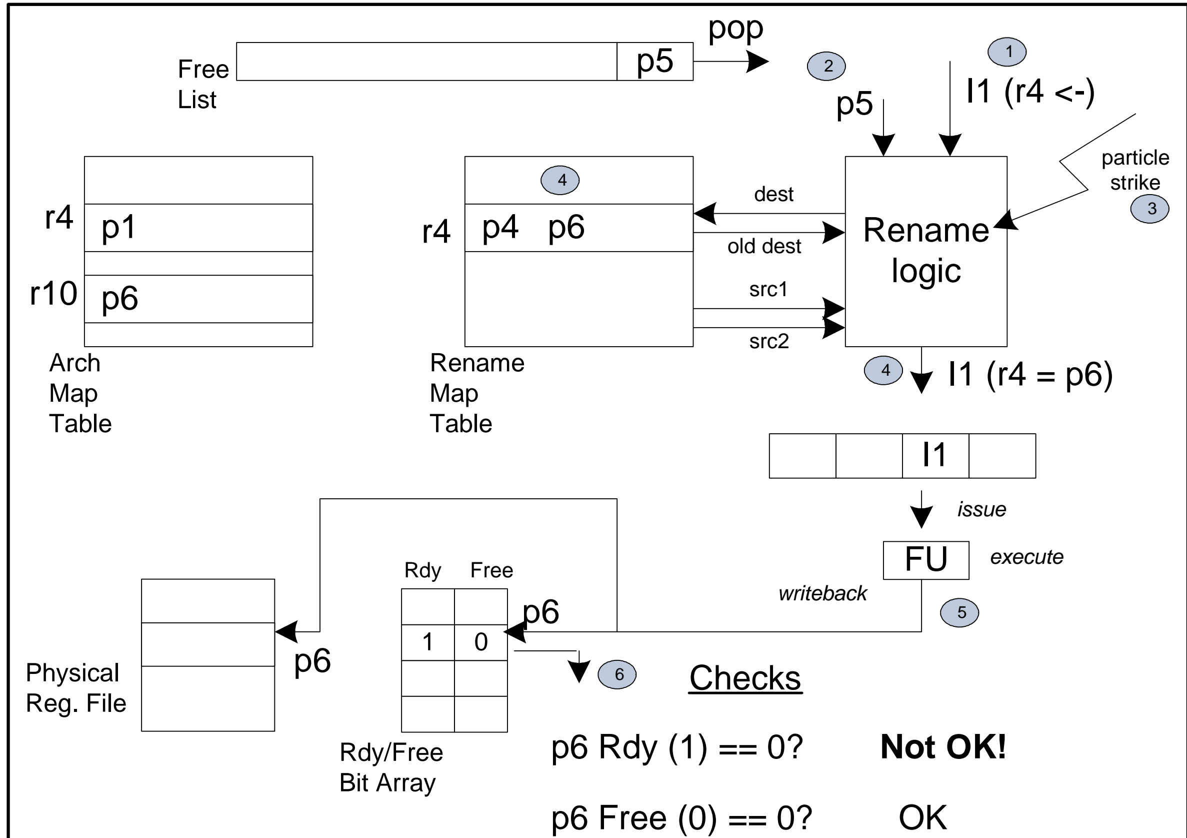


Figure 4-2. Fault detection using RNA-writeback.

The RNA writeback state check is illustrated in Figure 4-2. Again, various events are indicated by numbers in order of occurrence. At time 3, a fault causes the destination renaming logic to produce a wrong physical register tag p6 instead of p5, for instruction I1. P6 was last used by an instruction that finished and committed to the architectural state, and p6 is still part of the architectural state (shown by r10 – p6 in the architecture map table). Hence, p6 has {ready, free} of {1, 0}. After I1 executes and before it writes to p6, the ready and free bits of p6 are checked for consistency. At time 6, the check fails because the ready bit of p6 appears to have been set already. Hence, the fault is detected.

4.3 Source Register Renaming Faults

The two RNA checks discussed above can only detect faults on destination register mappings of instructions. However, RNA cannot detect faults purely on source register mappings, i.e., the faults on source register mappings that do not create any destination register inconsistencies. One way to protect source register mappings is to perform source renaming with the architecture map table at the commit stage of the pipeline, and confirm the correctness of the source register mappings. Since the architecture map table reflects the register mappings of all instructions prior to the committing instruction, the source register mappings obtained at the rename stage of the pipeline must match the source register mappings in the architecture map table. This fault check is implemented and evaluated in a later chapter. For some faults on source register mappings, a timeout mechanism like a watchdog timer (e.g., [1]), could be an effective assertion check. A watchdog timer can detect some pure source renaming errors, if they cause a timeout by blocking retirement, waiting for phantom producers to issue. Examples are faulty source registers in the free list that never get popped, faulty source registers in the forward slice of the faulty instruction that cause a cyclic dependency, etc.

4.4 Fault Injection

4.4.1 Experimental Setup

The RNA assertion checks were implemented in a cycle-level simulator to evaluate their fault detection capability. Faults are randomly injected into the microarchitectural state of the simulator pertaining to the rename and issue units.

The modeled microarchitecture is similar to the MIPS R10000 [41]. The microarchitecture configuration is shown in Table 4-1.

A separate, “golden” (fault-free) simulator is run in-sync with the faulty simulator. When an instruction is committed to the architectural state in the faulty simulator, it is compared with its golden counterpart to determine whether or not the architectural state is being corrupted. Any fault that leads to corruption of architectural state is classified as a potential silent data corruption (SDC) fault. Likewise, if no corruption of architectural state is observed for a set period of time after a fault is injected (the observation window), it is classified as a masked fault. In this study, an observation window of one million cycles is used. Results are similar for a five million cycle observation window.

Table 4-1. Microarchitecture configuration

L1 I & D Caches	64KB, 4-way, 64B line, LRU, L1hit = 1 cycle, L1miss/L2hit = 10 cycles
L2 Unified Cache	1MB, 8-way, 64B line, LRU, L1miss/L2miss = 100 cycles
Branch Predictor	gshare, 16-bit history, 2^{20} entries
Superscalar Core	reorder buffer (ROB): 64 dispatch/issue/retire bandwidth: 4 per cycle

As a preliminary test of RNA, the simulator is first run without fault injection. None of the assertion checks fire, indicating that there are no false alarms during correct operation.

Each fault injection campaign consists of 1,000 faults. The campaign is repeated for nine SPEC2K benchmarks. The benchmarks are compiled with the Simplescalar gcc compiler [3] for the PISA ISA.

A fault may lead to one of four possible outcomes, depending on (1) whether the fault is detected by an assert (“Assert”) or not (“Undet”) and (2) whether the fault corrupts

architectural state ("SDC") or not ("Masked"). Thus, the four possible outcomes of a fault are Assert+SDC, Assert+Masked, Undet+SDC, and Undet+Masked.

The combination of an assertion check and a SDC (Assert+SDC) occurring in the same observation window is interesting, because it indicates that the assertion check was able to detect a potential silent data corruption. The order of occurrence of the two events (Assert before SDC vs. SDC before Assert) is not of concern, as the focus is on fault detection. For RNA, either order may occur.

An assertion check may also detect a fault that is ultimately masked (Assert+Masked). For example, if a faulty register mapping in the rename map table is overwritten before being consumed by any instruction, the fault is detected by RNA but is masked.

As a final note about methodology, perfect branch prediction is modeled in order to minimize masking due to speculative state. This way, the chance that a fault surfaces as an error is increased, thus testing the fault detection capabilities of RNA better.

4.4.2 Results

To test RNA, faults are injected that cause renaming anomalies similar to those discussed in Section 4.1. In particular, four types of faults are injected, 1) bits of a random entry in the architectural map table are flipped (arch_map), 2) bits of a random entry in the rename map table are flipped (rename_map), 3) bits of a random entry in the physical register freelist are flipped (freelist) and 4) bits of the destination physical register tag of an instruction are flipped at dispatch, to emulate a destination renaming logic error (dest).

For the dest fault, only the instruction's tag is affected by the fault, i.e., the rename map table and future consumers get the correct tag. This creates a disconnect between the faulty producer and its consumers, potentially causing deadlock. To detect deadlocks that phantom producers can cause, a watchdog timer check (wdog) is included aside from the two RNA checks. Faults are discussed in relation to the checks that detect them (prevmapping, writeback, wdog), later in this section.

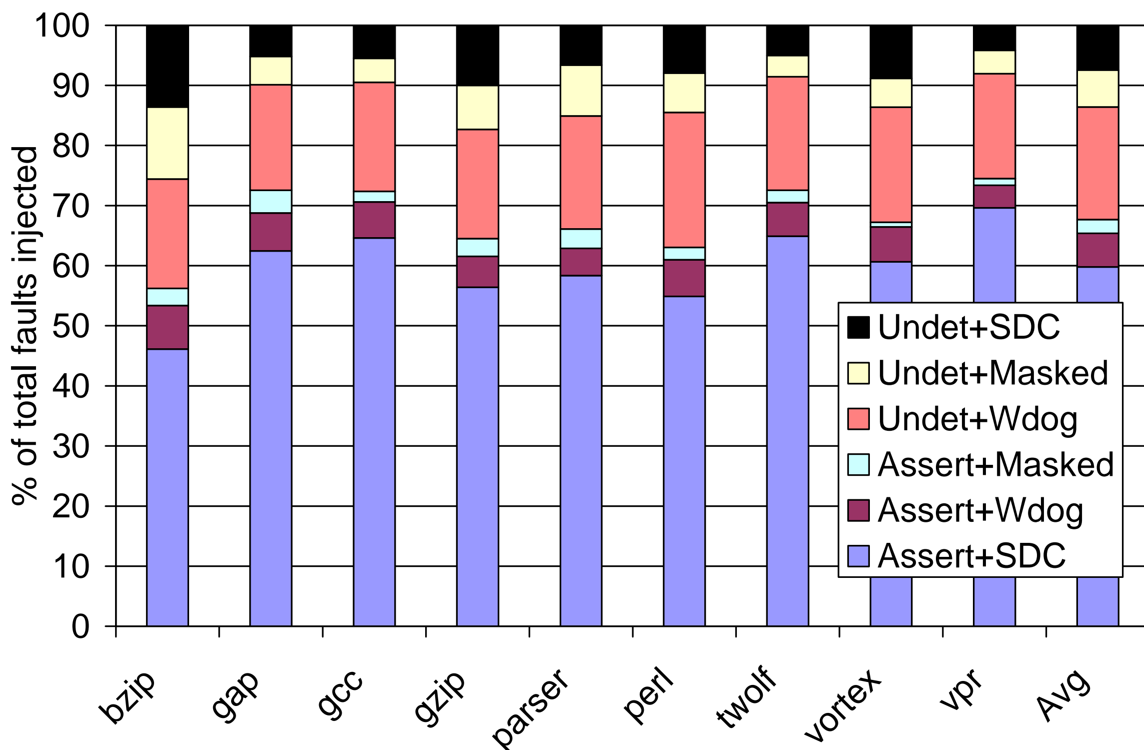


Figure 4-3. Breakdown of outcomes of RNA fault injection campaign.

A breakdown of fault outcomes is shown in Figure 4-3. Corresponding to the watchdog timer, two new outcomes appear in Figure 4-3. Assert+Wdog represents faults that are detected by RNA assertion checks, which would have later resulted in a deadlock had they not been detected. Undet+Wdog represent faults that escape RNA assertion checks and cause a deadlock, and hence will only be detected by a watchdog timer.

As shown in Figure 4-3, on average, 60% of the faults are detected by RNA assertion checks alone, that would otherwise cause a SDC (Assert+SDC). Another 6% of the faults are detected by RNA assertion checks, that would have caused deadlocks had there not been the earlier RNA checks (Assert+Wdog). Finally, another 2% of the faults are detected by RNA assertion checks, that are also masked (Assert+Masked).

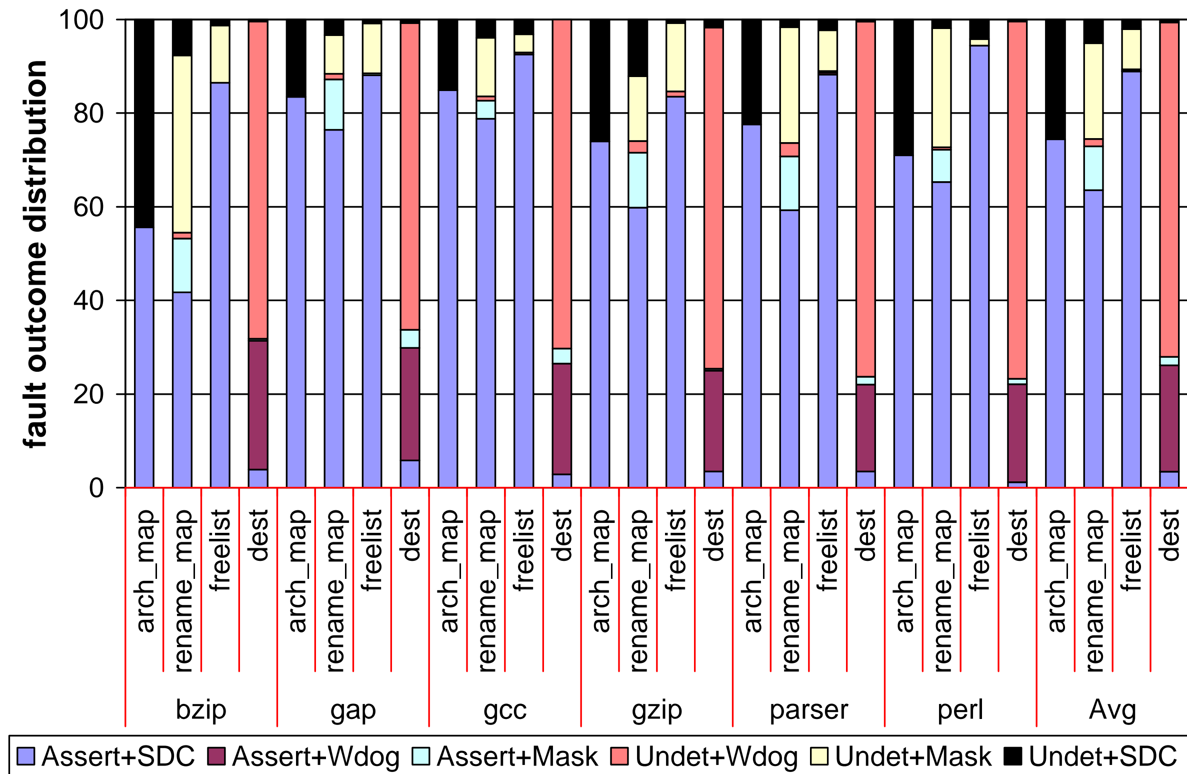


Figure 4-4. Relation of faults to outcomes.

About 18% of the faults escape RNA checks and cause a deadlock, requiring the watchdog timer to detect the faults (Undet+Wdog). About 8% of the faults lead to SDC and are not detected by the two RNA checks (Undet+SDC).

Hence, the two RNA checks proposed in this chapter provide reasonable fault detection – 67% of the injected faults are detected on average. Considering the low overheads of RNA, this is a very promising result.

To further understand the fault detection capability of RNA, and possibly enhance it, the relationship between different fault types and the corresponding outcomes they produce is investigated. Figure 4-4 shows the four fault types used in the RNA fault injection campaign on the x-axis (arch_map, rename_map, freelist, and dest) and the distribution of all possible outcomes due to a fault on the y-axis.

It is noticed that some faults get very good detection coverage. For instance, faults on freelist entries are detectable 90% of the time, on average. Also notice, majority of undetected, deadlock-causing faults are injected at dispatch (post-renaming) on destination physical register tags of instructions. Since the corresponding mapping in the rename map table is not faulty, consumers of the faulty instruction get phantom producers. A deadlock ensues, preventing the RNA previous mapping check from finishing, and instead causing the watchdog timer to fire.

A good result is that faults on the architectural map get detected more than 70% of the time, on average. At the same time, a majority of SDC also occurs due to faults on the architectural map. Faults on mappings of registers with long live ranges were found not to undergo an RNA check within the observation window (1 million cycles after fault). Another reason is occurrence of system trap before RNA detection. This motivates further schemes to protect the committed state.

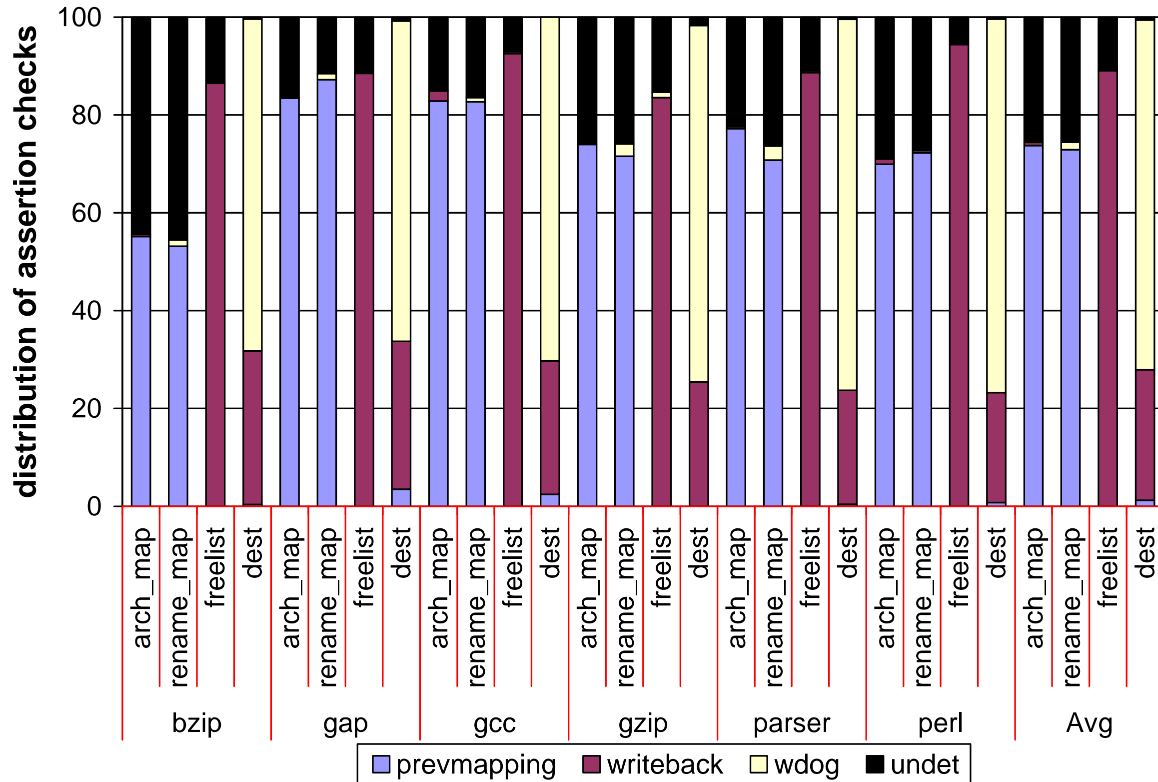


Figure 4-5. Relation of faults to assertion checks.

Next, the relation between fault types and different RNA assertion checks which detect them is investigated. This may enable understanding which schemes are more suitable for particular types of faults. Figure 4-5 shows the four fault types on the x-axis and their coverage by various assertion checks on the y-axis.

As expected, the prevmapping check is most suitable for faults on map table entries. For faults on the freelist, the writeback check provides most of the fault coverage. The prevmapping check fails since faults in the freelist are consistent in all tables. Writeback checks provide limited coverage of destination renaming logic faults, which are mainly covered by watchdog timer checks.

4.5 Summary

This chapter introduced Register Name Authentication (RNA) for detecting faults that affect destination physical register mappings. The two RNA checks, RNA previous mapping check and the RNA writeback state check, were found to provide substantial fault coverage to faults injected to register renaming related structures: architecture map table, rename map table, shadow map table, free list, and destination register mapping fields in the processor pipeline latches.

Chapter 5: Timestamp-based Assertion Checking

Timestamp-based Assertion Checking (TAC) provides fault coverage for the out-of-order scheduler in a superscalar processor. TAC reasons that an instruction executes only after all its producers have issued and executed. This time-orderliness within a data dependence chain is captured by assigning timestamps to instructions at issue. At retirement, an instruction's timestamp is compared with its producers' timestamps and an inconsistency indicates a violation in issuing order, and hence a fault in the scheduler.

When instructions issue to functional units in the wrong order due to a fault, it causes them to execute with wrong source values and produce a wrong result, ultimately leading to a wrong program outcome. TAC is a low-overhead assertion check that detects such faults. This chapter discusses TAC and evaluates its fault tolerance capabilities using fault injection.

The rest of the chapter is organized as follows. Section 5.1 discusses the working of TAC. TAC uses counters for timestamps, and since counters can overflow, it could lead to incorrect fault detection by TAC. Section 5.2 discusses how timestamp counter overflows are handled. Section 5.3 evaluates the fault tolerance capabilities of TAC using fault injection. Section 5.4 summarizes the chapter.

5.1 TAC Mechanism

To illustrate how TAC works, consider the instruction sequence (and the corresponding data-flow graph) shown in Figure 5-1. Frames A to F in Figure 5-2 show how TAC checks the issue logic by assigning timestamps to these instructions.

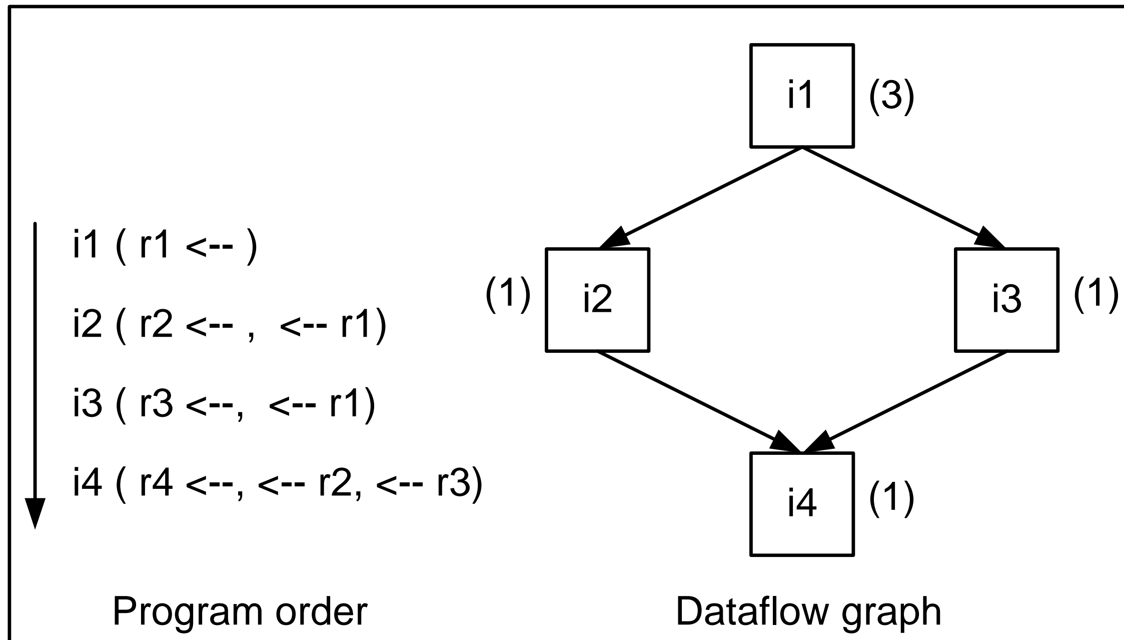


Figure 5-1. Example data-flow graph with latencies.

Boxes in dark grey indicate new additions to the existing microarchitecture to support TAC. The new additions are: 1) a timestamp counter incremented each cycle and 2) fields in the ROB and architecture map table to store timestamp and execution latency information.

In frame A, all instructions have been fetched and renamed into the ROB. For convenience, the timestamp counter is assumed to be at 1 at this point. In frame B, all instructions have issued. The timestamps recorded at issue are shown in their ROB entries. In frames C through F, the instructions are committed. When a given instruction I commits, the following main consistency check is performed with its two producers S1 and S2 (the TAC-LAT check):

$$I_ts >= S1_ts + S1_lat$$

AND

$$I_ts >= S2_ts + S2_lat$$

Here, I_ts is the timestamp of instruction I and I_lat is its latency (same for S1 and S2). A slightly less rigorous check is possible at a lower implementation cost (the TAC-TS check):

$I_ts > S1_ts$ AND $I_ts > S2_ts$.

This consistency check determines whether or not the issue order of instruction I was correct with respect to its producers. However, it is less rigorous, because even if the issue order of instruction I was correct with respect to its producers, it may be that instruction I read its source operands before its producers finished executing. The TAC-LAT check detects even such premature issuing errors, and hence subsumes the TAC-TS check.

In frame C, i1 commits and, as shown, both consistency checks confirm it issued correctly. Frames D and E confirm the same for i2 and i3, respectively. In frame F, since i4 has two sources, consistency checks are performed with respect to both of them, which also succeed.

If i4 had issued with a {timestamp, latency} of {5, 1}, then in frame F, either of the TAC check would compare timestamps and fail (specifically, $5 > 6$ and $5 > (6+1)$), detecting an issue order violation (i4 issuing before i2). Similarly, if i3 issued with a {timestamp,

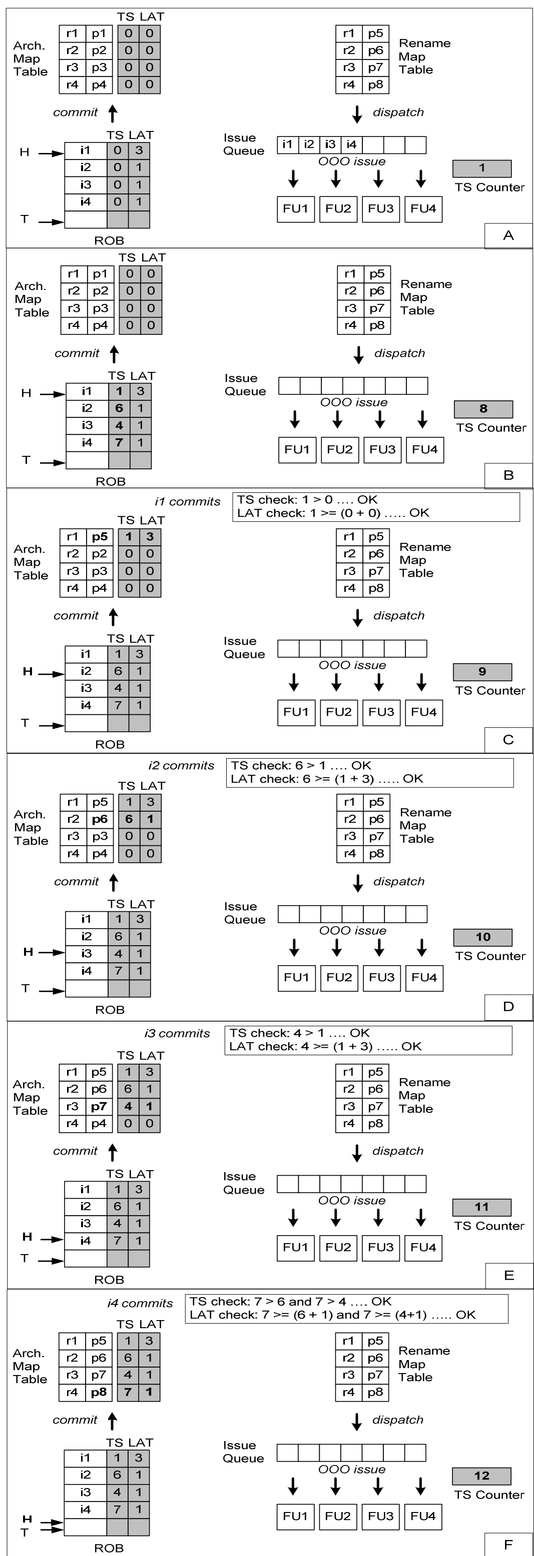


Figure 5-2. Illustration of TAC checks.

latency} of {3, 1}, then in frame E, the TAC-TS check would succeed (because i3 issued after i1) but the TAC-LAT check would fail (because $3 < (3 + 1)$) and detect a premature issuing violation.

5.2 Handling Wraparound of Timestamp Counter

TAC uses a timestamp counter incremented each cycle. When the counter wraps around, it may cause false error alarms. Hence, the wraparound case needs special handling.

To deal with wraparounds, each new counter wraparound is viewed as entering a new phase. If the timestamp counter is big enough that only one wraparound can occur within the scope of the ROB, we can be assured each committing instruction has a timestamp from either the current phase or the previous phase (i.e., immediately before the current phase). To identify the current phase, a toggling current phase bit is maintained along with the timestamp counter. A wraparound toggles the current phase bit. To remember the phase that an instruction's timestamp belongs to, a phase bit is also appended to each ROB entry.

At commit, an instruction's phase can be identified by comparing its phase bit with the current phase bit: if equal, it issued in the current phase, else it issued in the previous phase.

To identify the phase of an instruction's producer, two bits are added to the architecture map table – a current-phase production bit and a previous-phase production bit. When a producer instruction commits, it sets either the current-phase production bit or previous-phase production bit corresponding to its logical destination register, depending on the phase of its timestamp. At counter wraparound, current-phase production bits are flash copied into the previous-phase production bits (and then all current-phase production bits are

reset). If both bits are unset, it indicates a past-phase (i.e., neither current phase nor previous phase) production.

At commit, a current-phase instruction is compared only with its current-phase producers, i.e., comparisons with previous-phase or past-phase producers are skipped to avoid false alarms. Moreover, skipping previous-phase and past-phase timestamp comparisons is safe, because the fact that previous-phase and past-phase producers are identified confirms that the current-phase consumer instruction issued after these producers (implicitly implements the TAC-TS check without timestamp comparisons). There is a slight residual vulnerability since the latencies of previous-phase and past-phase producers are not accounted for (no implicit TAC-LAT check). Similarly, at commit, a previous-phase instruction is compared only with previous-phase producers, i.e., comparisons with past-phase producers are skipped using the same reasoning as before. Moreover, the previous-phase instruction is in issue order violation if it has any current-phase producers.

A sufficiently large timestamp counter enables handling counter wraparounds elegantly and prevents false alarms, as discussed above. For a 64-entry ROB and a L2-miss latency of 100 cycles, a 13-bit counter is sufficient to ensure at most one wraparound within the scope of the ROB, in the worst case of chained L2 misses.

5.3 Fault Injection

5.3.1 Experimental Setup

The TAC assertion checks were implemented in a cycle-level simulator to evaluate their fault detection capability. Faults are randomly injected into the microarchitectural state of the simulator pertaining to the rename and issue units.

The modeled microarchitecture is similar to the MIPS R10000 [41]. The microarchitecture configuration is shown in Table 5-1.

A separate, "golden" (fault-free) simulator is run in-sync with the faulty simulator. When an instruction is committed to the architectural state in the faulty simulator, it is compared with its golden counterpart to determine whether or not the architectural state is being corrupted. Any fault that leads to corruption of architectural state is classified as a potential silent data corruption (SDC) fault. Likewise, if no corruption of architectural state is observed for a set period of time after a fault is injected (the observation window), it is classified as a masked fault. In this study, an observation window of one million cycles is used. Results are similar for a five million cycle observation window.

Table 5-1. Microarchitecture configuration

L1 I & D Caches	64KB, 4-way, 64B line, LRU, L1hit = 1 cycle, L1miss/L2hit = 10 cycles
L2 Unified Cache	1MB, 8-way, 64B line, LRU, L1miss/L2miss = 100 cycles
Branch Predictor	gshare, 16-bit history, 2^{20} entries
Superscalar Core	reorder buffer (ROB): 64 dispatch/issue/retire bandwidth: 4 per cycle

As a preliminary test of RNA, the simulator is first run without fault injection. None of the assertion checks fire, indicating that there are no false alarms during correct operation.

Each fault injection campaign consists of 1,000 faults. The campaign is repeated for nine SPEC2K benchmarks. The benchmarks are compiled with the Simplescalar gcc compiler [3] for the PISA ISA.

A fault may lead to one of four possible outcomes, depending on (1) whether the fault is detected by an assert ("Assert") or not ("Undet") and (2) whether the fault corrupts

architectural state (“SDC”) or not (“Masked”). Thus, the four possible outcomes of a fault are Assert+SDC, Assert+Masked, Undet+SDC, and Undet+Masked.

The combination of an assertion check and a SDC (Assert+SDC) occurring in the same observation window is interesting, because it indicates that the assertion check was able to detect a potential silent data corruption. The order of occurrence of the two events (Assert before SDC vs. SDC before Assert) is not of concern, as the focus is on fault detection. TAC always fires an assertion check before the detected fault can cause a SDC, permitting safe recovery from the architectural state.

An assertion check may also detect a fault that is ultimately masked (Assert+Masked). For example, if a faulty register mapping in the rename map table is overwritten before being consumed by any instruction, the fault is detected by TAC but is masked.

As a final note about methodology, perfect branch prediction is modeled in order to minimize masking due to speculative state. This way, the chance that a fault surfaces as an error is increased, thus testing the fault detection capabilities of TAC better.

5.3.2 Results

To test TAC, faults that cause instructions to issue prematurely are randomly injected. In particular, (1) ready bits of physical registers are prematurely set and (2) speculatively issued dependents of cache-missing loads are not reissued. The TAC-LAT check discussed in Section 5.1 is used to detect faults. Results are shown in Figure 5-3.

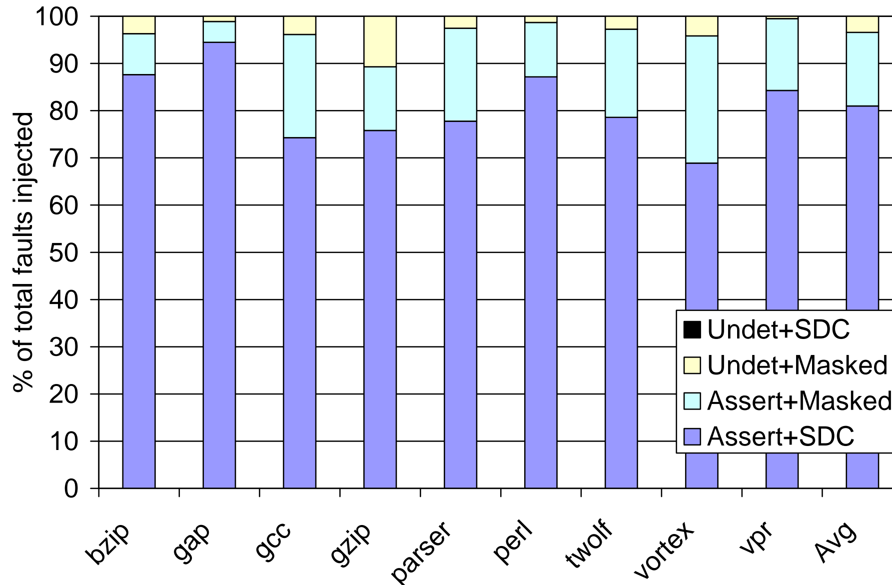


Figure 5-3. Breakdown of outcomes of TAC fault injection campaign.

As seen in Figure 5-3, on average, 80% of the faults are detected by TAC, that would otherwise cause a SDC (Assert+SDC). Another 17% of the faults are detected by TAC, that are masked (a prematurely issued instruction produces the correct outcome even with wrong operands). The remaining 3% of the faults are undetected and masked, because the faults did not cause instructions to issue prematurely (e.g., select logic may stall issuing the instructions long enough). Notice that none of the faults were allowed to cause a SDC (no Undet+SDC). This is expected because TAC detects violations before wrongly issued instructions commit to the architectural state.

5.4 Summary

This chapter introduced Timestamp-based Assertion Checking (TAC) for detecting faults that cause instructions to issue in the wrong order. TAC mainly detects faults in the issue unit of a processor comprising of complex hardware schedulers that track instruction data dependencies, and complex logic to wakeup and select correct instructions for issuing. A

transient fault in the scheduler or any associated logic involved can cause instructions to misfire. TAC detects all of these faults with one simple assertion check. Since a fault is detected at retirement, successful recovery is done by simply flushing the pipeline and restarting from the faulty instruction.

Chapter 6: A Canonical Fault-Tolerant Superscalar Processor

This chapter develops and evaluates a canonical fault-tolerant superscalar processor. A regimen comprising of the fault checks developed in this dissertation is incorporated into a superscalar processor pipeline. To evaluate fault tolerance of the processor, detailed fault injection is performed on a timing simulator. To testify my second thesis, it is shown that the regimen of canonical fault checks developed in this dissertation provides substantial fault coverage to the processor.

The rest of the chapter is organized as follows. Section 6.1 discusses the fault injection strategy used in this study to evaluate fault tolerance of the regimen-based processor design. Section 6.2 presents a detailed processor pipeline model, based on the IVM [61], and analyzes each pipeline stage to determine the possible faults, culminating with a list of faults for injection into a timing-based simulator. Section 6.3 presents the fault-checking regimen and highlights the parts of the pipeline protected by each fault check. Section 6.4 explains the experimental methodology and Section 6.5 presents results for the fault injection experiments.

6.1 Fault Injection Strategy

Fault injection in this study aims to model the anomalies caused by natural radiation interference in a superscalar processor pipeline, and study the response of the fault-checking regimen in detecting the anomalies. Fundamentally, a particle strike can flip a stored bit in a

sequential element (like a latch or flip-flop) or cause a transient pulse in a net that might lead to incorrect outputs from combinational logic blocks. Modeling these fundamental faults requires a gate-level implementation of a processor pipeline. Though this method is highly accurate, for multiple reasons, fault injection on a more abstract model of the processor pipeline might be desirable:

1. In early stages of processor development, designers only have access to abstract behavioral models like timing simulators. Since designing a fault-checking regimen is part of the processor development cycle, estimates about reliability is required early on to make improvements and tradeoffs. Thus, it is desirable to have a reasonably accurate fault injection strategy that could be used on a timing simulator.

2. Fault simulation at the gate-level is time consuming. It is not suitable for use during early design phases, when several alternatives need to be considered and fairly accurate, but fast, evaluation is important. This demand can be met if it were possible to obtain reasonable fault tolerance estimates from timing simulators.

3. Fault injection at the gate-level has high levels of inherent fault masking [32]. This could be due to (i) architecture-level masking, where some fault-afflicted bits are insignificant for correct architectural execution [10], or (ii) logic-level masking, where a fault-afflicted bit does not change the outputs of a logic block, or (ii) circuit-level masking, where a blip in a net is not latched, or returns to equilibrium before getting latched [33][34]. While inherent fault

masking should be part of an accurate reliability estimate, it is not useful in evaluating the effectiveness of a fault-checking regimen. A more fitting approach to evaluate the fault-checking regimen is to directly model anomalies caused by lower-level faults at a behavioral level, like in a timing simulator. Then, the incidence of unmasked faults would be higher, resulting in a rigorous evaluation of the fault-checking regimen.

The downside of using a high-level timing simulator for fault injection is uncertainty regarding the actual pipeline coverage achieved by the fault injection campaigns. To overcome this discrepancy, and enable rigorous fault tolerance evaluation in a timing simulator, a new approach is proposed in this dissertation. The idea is to aggregate faults at several locations in the microarchitecture of a processor pipeline stage into visible fault manifestations that can be modeled in a timing simulator. A visible fault is, then, representative of several lower-level faults caused by particle strikes. By analyzing each pipeline stage and mapping out the visible faults, a rigorous fault injection campaign is achieved to evaluate the fault-checking regimen, while not significantly sacrificing pipeline coverage to a great degree.

There have been other studies that provide reliability estimates at an early processor design phase [31][10]. Hierarchical simulation [31] is closely related to the fault injection strategy used in this study. With hierarchical simulation, the effects of low-level transient faults are propagated to higher levels of abstraction using fault dictionaries, allowing fault-injection-based studies at any desired level, for example, system level. The fault aggregation approach used in this study could be viewed as propagating effects of faults at a gate level to

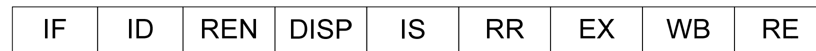
a microarchitecture level. Our methodology is novel in its contribution to analyzing a detailed superscalar processor pipeline. AVF analysis [10][11] also enables reliability estimation using timing simulators. However, AVF is only suitable for providing vulnerability estimates of stateful structures like the issue queue, and is incapable of estimating the vulnerability of combinational logic, which is a significant portion of processor pipelines. Additionally, though AVF analysis is capable of estimating vulnerability of stateful structures, it is incompatible with the fault-checking regimen used in this study. This is because the fault checks in the regimen are based on behavioral insights into the microarchitecture, and their detection capability depends upon a combination of various microarchitecture events. Keeping track of microarchitecture events and their combinations each clock cycle, to adjudge whether or not bits in a structure are protected by the fault-checking regimen, is an intractable problem. For this reason, random fault injection is the most suitable approach to evaluate the fault tolerance of a processor pipeline augmented with the fault-checking regimen.

Section 6.2 presents a detailed pipeline model for a superscalar processor. Each pipeline stage is thoroughly inspected to map out the visible faults. The outcome of this exercise is a comprehensive list of visible faults that is representative of several faults in each pipeline stage. The faults are randomly injected into a timing simulator to evaluate the fault tolerance capabilities of the regimen-based processor.

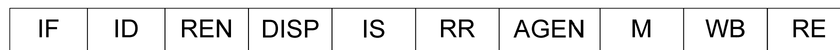
6.2 Fault Analysis of a Superscalar Pipeline

A typical superscalar processor pipeline comprises of several stages that process instructions in parallel, as depicted in Figure 6-1. The instruction fetch (IF) stage predicts

branches and fetches instructions from the instruction cache. The decode stage (ID) decodes instructions to generate decode signals for processing the instruction. The rename stage (REN) renames instructions to eliminate false dependencies in the static program representation. The dispatch stage (DISP) inserts instructions into the reservation stations (i.e., the issue queue) and the reorder buffer, and also load/store buffers, in the case of memory instructions. The issue stage (IS) schedules instructions for execution based on data availability and issue bandwidth. After being issued, instructions read source values from the register file in the register read stage (RR), and begin executing in the execution stage (EX). Loads and stores go through the address generation (AGEN) stage instead of the EX stage, followed by the disambiguation and data cache access stage (M). After execution or data cache access, instructions write their results in the register file and bypass them to dependent instructions in the writeback stage (WB). Instructions are finally retired in original program order from the reorder buffer in the retirement stage (RE).



(a)



(b)

Figure 6-1. Superscalar processor pipeline for (a) non-memory and (b) memory instructions.

In the following subsections, fault analysis is performed on each of the pipeline stages of a two-way superscalar processor. A visible fault is represented by a solid circle with a number associated with the fault, for example, ❶. As discussed in Section 6.1, a visible fault captures the effect of several lower-level faults, and can be represented as a fault in the timing simulator. In some cases, a visible fault might be redundant with other visible faults in

the pipeline, i.e., the effects of two or more visible faults might be the same, and could conceivably be captured by just one visible fault in the pipeline. In some cases, the redundant visible faults are aggregated into one visible fault. Reducing the number of faults modeled saves valuable modeling effort, and removing redundant faults from the test space makes the fault injection campaign more effective. In the pipeline analysis in the subsections that follow, a redundant visible fault that is aggregated by another visible fault, for example, ❶, is indicated by an empty circle with the fault number of the proxy visible fault, i.e., ①. A redundant visible fault that is aggregated by a visible fault in another pipeline stage, for example, ❶ in the rename stage, is indicated by an empty circle with the fault number of the aggregated visible fault prefixed by the abbreviation for the pipeline stage, i.e., (ren1).

Each of the following subsections presents a detailed description of a given pipeline stage and analyzes it for faults: the fetch and decode stages are discussed in Section 6.2.1, the register rename stage is discussed in Section 6.2.2, the dispatch stage is discussed in Section 6.2.3, and the backend stages of the pipeline (issue, register read, execute, register writeback, data cache access, and retire) are discussed in Section 6.2.4.

6.2.1 Fetch and Decode Stages

6.2.1.1 Description

Figure 6-2 is a detailed description of the fetch and decode stages of a superscalar processor pipeline. There are three fetch stages, Fetch0, Fetch1 and Fetch2. In Fetch0, the instruction cache unit is accessed using the PC (program counter). The instruction cache unit (ICU) comprises of the instruction translation lookaside buffer (I-TLB), instruction cache, and instruction alignment logic. The outputs of the ICU are a maximum of two instructions,

which is the fetch bandwidth of the processor. In some cases, the output could be only one instruction. To signify the validity, instructions are marked with valid bits (v1 and v2). In parallel with accessing the ICU, the PC is also fed to the Next PC Prediction unit for predicting the PC for the next cycle. The prediction is based on inputs from a branch target buffer (BTB), branch predictor and return address stack (not explicitly shown in the figure). The predicted next PC is fed back as a candidate PC for the next cycle.

In the Fetch1 stage, it is checked whether or not there has been a misfetch due to wrong branch information from the Next PC Prediction Unit. Inputs to Fetch1 are the instruction packet fetched from the Fetch0 stage, BTB information about branches in the packet that was assumed by the Next PC Prediction Unit to predict the next PC (in the case of a BTB miss, the Next PC Prediction Unit assumes that the fetched packet contains no branches), and the predicted next PC itself. The fetched packet is decoded to extract branch information and compared with the BTB information used by the Next PC Prediction Unit. The following inconsistencies are possible: 1) a branch is not present in the packet whereas it was predicted to be present by the BTB, 2) a branch is present in the packet whereas it was predicted not to be present by the BTB (a BTB miss falls in this category), 3) a branch is present, as predicted, but its position in the packet is predicted incorrectly by the BTB, and 4) for predicted taken branches, the predicted target address from the BTB is inconsistent with the decoded target address. On detecting an inconsistency, the Fetch0 stage is redirected to the override PC, and instructions in the interim are flushed. The Next PC Prediction Unit is also updated with the correct information (not shown in figure).

In the Fetch2 stage, the instructions fetched from Fetch1 are entered into a fetch queue. The fetch queue decouples the decode unit from the fetch unit. The FetchQ Allocator tracks the number of entries in the fetch queue, and if space is available, enters the new instructions at the tail of the queue (tail and tail+1). The FetchQ Deallocator does just the opposite – it reads instructions from the head of the fetch queue (head and head+1) and provides it to the Decode stage.

In the decode stage, instructions are fully decoded to establish signals that govern the processing of the instruction in later pipeline stages.

6.2.1.2 Faults

❶ Fault in the program counter (PC). Three types of faults are possible, 1) a bit-flip in the PC or a bit-flip in any of the possible PCs which are inputs to the PC multiplexer, 2) selection of a wrong input PC from the PC multiplexer, and 3) a different branch target due to selection of a wrong target address by the next PC prediction unit.

❷ Fault in the fetched instructions. Two types of faults are possible, 1) a bit-flip in an instruction, and 2) fetching of unintended or faulty instructions due to a fault in the instruction cache unit (even though the PC was correct), or a fault in the flush signal from the Fetch1 stage, causing incorrect instructions to remain in the pipeline.

❸ ❹ Fault on the valid bits leading to appearance of new instructions or disappearance of instructions.

❺ A bit-flip in an instruction in the FetchQ.

❻ Instructions overwriting other instructions in the fetch queue due to 1) a fault in the FetchQ Allocator, or 2) a fault on the ‘fetch queue full’ stall signal.

❼ Wrong instructions being read out of the fetch queue due to a fault in the FetchQ Deallocator.

❽ Fault in the decoded signals. Two types of faults are possible, 1) a fault in the decoder, and 2) a bit-flip in a decoded signal.

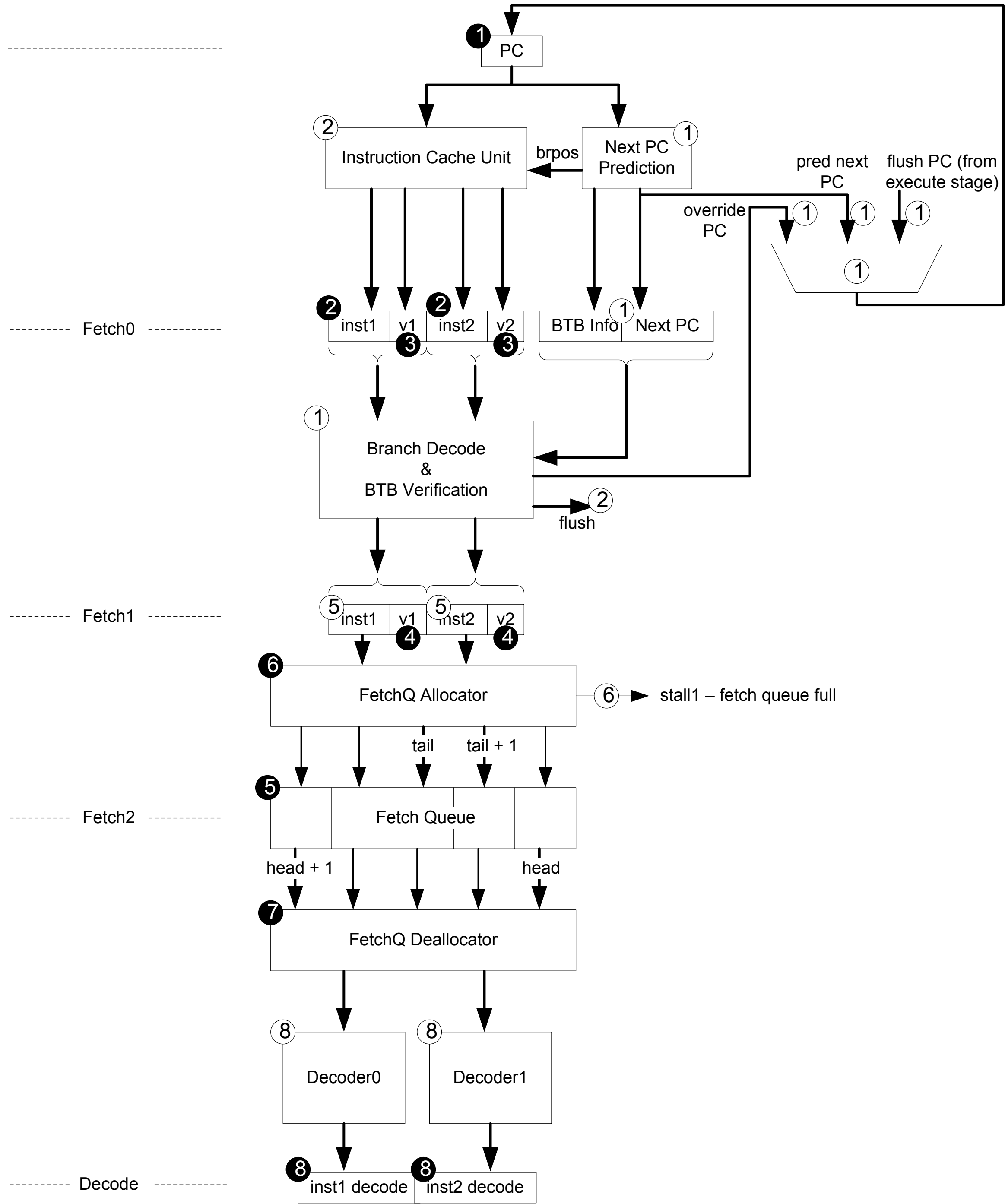


Figure 6-2. Fault analysis of the fetch and decode stages of the pipeline.

6.2.2 Register Rename Stage

6.2.2.1 Description

Figure 6-3 shows the register rename stage of the pipeline. All instructions access the rename map table to get their source physical register mappings. Instructions that have a destination operand obtain a physical register from the free list and update the rename map table with the new mapping. Before updating a mapping, an instruction reads and stores the old physical register mapping. This simplifies freeing of registers at instruction retirement. Since multiple instructions are renamed simultaneously (in this case, up to two instructions), there could be register conflicts between the renamed instructions, i.e., an earlier instruction could write to the destination register consumed by a later instruction in the packet. In this case, the source physical register mappings of the later instruction obtained from the rename map table are stale, and need to be updated with the destination physical register of the earlier instruction in the packet. The dependence checking logic detects such dependencies between instructions in a packet and generates select signals, used by mapping override multiplexers to choose between physical register mappings from the rename map table and destination physical registers of earlier instructions. Conditional branch instructions make a checkpoint of the rename map table for quick recovery from branch mispredictions.

6.2.2.2 Faults

❶ Erroneous physical register mapping in the rename map table. Three types of faults are possible, 1) bit-flip in a physical register mapping in the rename map table, 2) bit-flip in the physical register mapping in a branch checkpoint, which is then copied to the rename map table, and 3) restoring a branch checkpoint incorrectly.

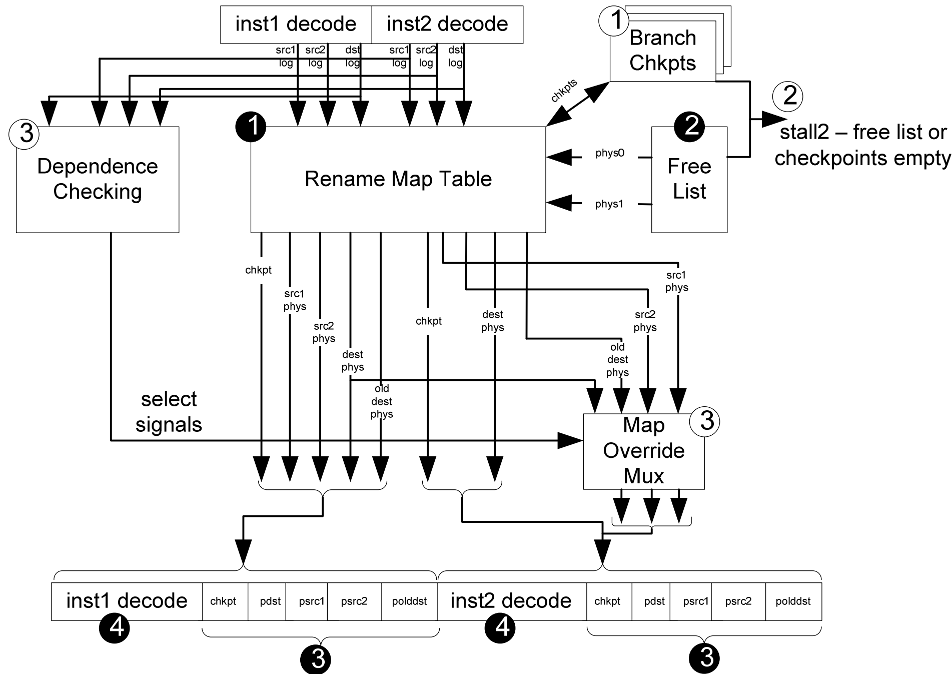


Figure 6-3. Fault analysis of the register rename stage of the pipeline.

❷ Fault in the physical register mappings given by the free list. This may happen due to, 1) a bit-flip in a physical register returned by the free list, or 2) a fault in the internal state of the free list causing it to return physical registers already in use by other instructions, or 3) a fault on the 'free list empty' stall signal.

❸ Fault in the outputs. Two types of faults are possible, 1) a bit-flip in an output latch (chkpt, P_{dst}, P_{src1}, P_{src2}, and P_{olddst}), and 2) a stale mapping due to a fault in either the dependence checking unit or the mapping override unit.

❹ A bit-flip in a decode signal.

6.2.3 Dispatch Stage

6.2.3.1 Description

Figure 6-4 shows the dispatch stage of the pipeline. All instructions get dispatched to two buffers: the reorder buffer (ROB) and the reservation stations (RS). Additionally, memory instructions get dispatched to either load or store buffers depending on whether they are load instructions or store instructions, respectively. The allocator tracks availability in all

the three buffers. If space is available in all the buffers an instruction is destined for, write ports are allocated and the instruction is written to the corresponding buffers. If space is not available, a stall is indicated and instructions wait until space is available. Instructions entering the RS read the ready bits for their source operands to check if they are ready for issue (explained in Section 6.2.4).

6.2.3.2 Faults

❶ A fault in the ROB write ports or a ROB bookkeeping fault in the allocator, that may lead to overwriting of valid entries in the ROB. This not only causes the disappearance of another instruction, but causes the dispatched instruction to be out-of-order in the ROB, impacting in-order commit and freeing.

❷ A fault on the 'no resource' signal that may lead to overwriting of valid entries in the RS or the ROB or the load/store buffers.

❸ A fault in the load or store buffer write ports, or a load or store buffer bookkeeping fault in the allocator, that may lead to overwriting of valid entries in the load or store buffers. This not only causes the disappearance of an another load or store, but may result in flawed memory forwarding and memory disambiguation later on.

❹ A fault in the RS write ports or a RS bookkeeping fault in the allocator, that may lead to overwriting of valid entries in the RS. This causes the disappearance of an instruction in the RS, which in turn impacts issuing of dependents, etc.

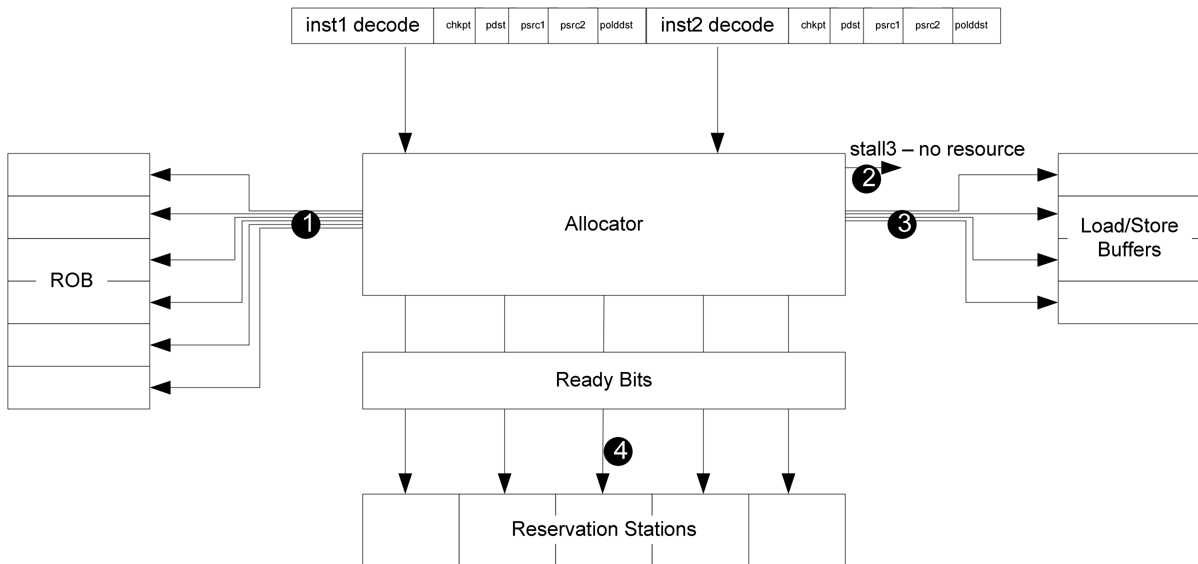


Figure 6-4. Fault analysis of the dispatch stage of the pipeline.

6.2.4 Backend Stages

6.2.4.1 Description

Figure 6-5 shows the backend stages (issue, register read, writeback, data cache access, and retire) of the processor pipeline.

The issue stage comprises of the RS and the wakeup/select logic. Instructions enter the RS after reading the ready bits of their source registers at dispatch. Some source operands may not have been produced yet, and their producers may still be in the process of being issued and executed. Instructions wait in the RS until their pending source operands are signaled to be ready by the wakeup logic. The wakeup logic broadcasts to the RS the physical tags of registers that will be ready (corresponding to just-issued instructions), and each instruction in the RS compares its pending source tags to the broadcasted tags to determine if it is ready for issue. When all source operands are ready, an instruction becomes a candidate for being issued, and signals its candidacy by sending a request (req) signal to the select logic.

Each cycle, the select logic receives request signals from all the instructions in the RS that are ready to issue in that cycle, and selects instructions based on available issue slots, functional units, and heuristics like age, etc, [72]. The selected instructions are informed through grant (gnt) signals, and subsequently issue. The select logic also schedules destination register tags of the selected instructions to be broadcasted via the wakeup ports. Scheduling is done based on the execution latency to allow for back-to-back execution of dependent instructions.

Issued instructions read their source operands from the register file in the register read stage and enter functional units for execution. Source operands read from the register file might be stale if they are being produced in the same cycle and, hence, are not yet available in the register file. To accommodate this, results (along with their physical register tags) produced in the execute stage are bypassed to instructions entering the functional units. After register read, if an instruction's source register matches a bypassed destination register tag, the value supplied from the register file is overridden with the bypassed value. This is key to enabling back-to-back execution of dependent instructions.

After execution, non-memory instructions writeback their results to the register file in the writeback stage. If there was a branch misprediction, a flush signal is sent to the fetch stage along with the flush PC to restart the pipeline. Instructions then set the complete bit in the ROB to indicate their completion status.

After address generation, load and store instructions update the load or store buffer and check for memory dependences by comparing this addresses with store or load address cams, respectively.

A load instruction updates the load buffer with its address and compares its address with prior store buffer entries to check for address conflicts. If a matching address is found in the store buffer, the youngest prior matching store forwards its data to the load. Otherwise, a data request is placed for the load with the data cache unit (DCU). When data becomes available in the DCU, it is forwarded to the load buffer. In either case, the data is then broadcasted over the result bypass and written back to the register file.

A store instruction updates the store buffer with its address and data, and disambiguates with all the load buffer entries logically after it to check for load violations. If a matching address is found in a load buffer entry, and the load has already received data either forwarded from either an older store or the DCU, an exception is marked for that load in the ROB. If the load has already received data from a younger store, nothing is done. If the load has not received any data yet, the store forwards its data to the load and pending DCU transfers to the forwarded load entry are then canceled.

The retirement logic polls the complete bits of instructions at the head of the ROB. Instructions that have completed are committed to architectural state. All instructions update the architectural map table with their destination physical registers, and free previously mapped physical registers back to the free list. Stores are committed by writing their data to the data cache from the store buffer. Any load exceptions, or other exceptions (like TLB misses, arithmetic exceptions, etc.) are handled at commit by taking the appropriate action. For load exceptions, the pipeline is flushed and restarted from the violating load.

6.2.4.2 Faults

❶ A bit-flip in a wakeup tag, that might cause several instructions in the RS to erroneously become ready for issue and prevent true dependent instructions from issuing.

❷ A bit-flip in the checkpoint (chkpt) field used by conditional branch instructions, that might cause the processor to roll back to the wrong checkpoint if the branch was mispredicted, or if correctly predicted, might cause a checkpoint being used by another branch to be released, not to mention not releasing the branch's checkpoint (resource "leak").

❸ A bit-flip in the ROB index where the instruction was dispatched, that might cause the instruction to update the wrong ROB entry.

❹ A bit-flip in the load buffer index used by load instructions, that might cause the calculated address to be placed in the wrong load buffer entry, and also lead to incorrect memory disambiguation, store-load forwarding, etc.

❺ A bit-flip in the store buffer index used by store instructions, that might cause the calculated address to be placed in the wrong store buffer entry, and also causing incorrect memory disambiguation, store-load forwarding, etc.

❻ A bit-flip in the grant signal that might cause a wrong RS entry to be released to the RS free list, and hence lead to overwriting of that RS entry with a newly dispatched instruction.

❼ A faulty source value, due to either 1) a bit-flip in the source value, or 2) a wrong match or non-match with a faulty physical tag broadcasted over the result bypass.

❽ A faulty destination value, due to either 1) a fault in the function unit, or 2) a bit-flip in the destination value.

❾ A fault on the calculated address that can cause a memory read from the wrong address

(for loads) or a memory write to a wrong address (for stores).

❿ A fault on the store value that can cause an incorrect value to be written to memory.

⓫ A fault on the complete bit in the reorder buffer, either due to an instruction updating a wrong entry in the ROB, or a bit-flip.

⓬ A fault on the instruction payload in the ROB (like source registers, destination registers, PC, etc), either due to a bit-flip, or an instruction updating the wrong reorder buffer entry.

⓭ A bit-flip in a register mapping in the architectural map table.

⓮ Wrong data being returned by the data cache unit, either due to a fault in the data cache unit or a bit-flip in the data.

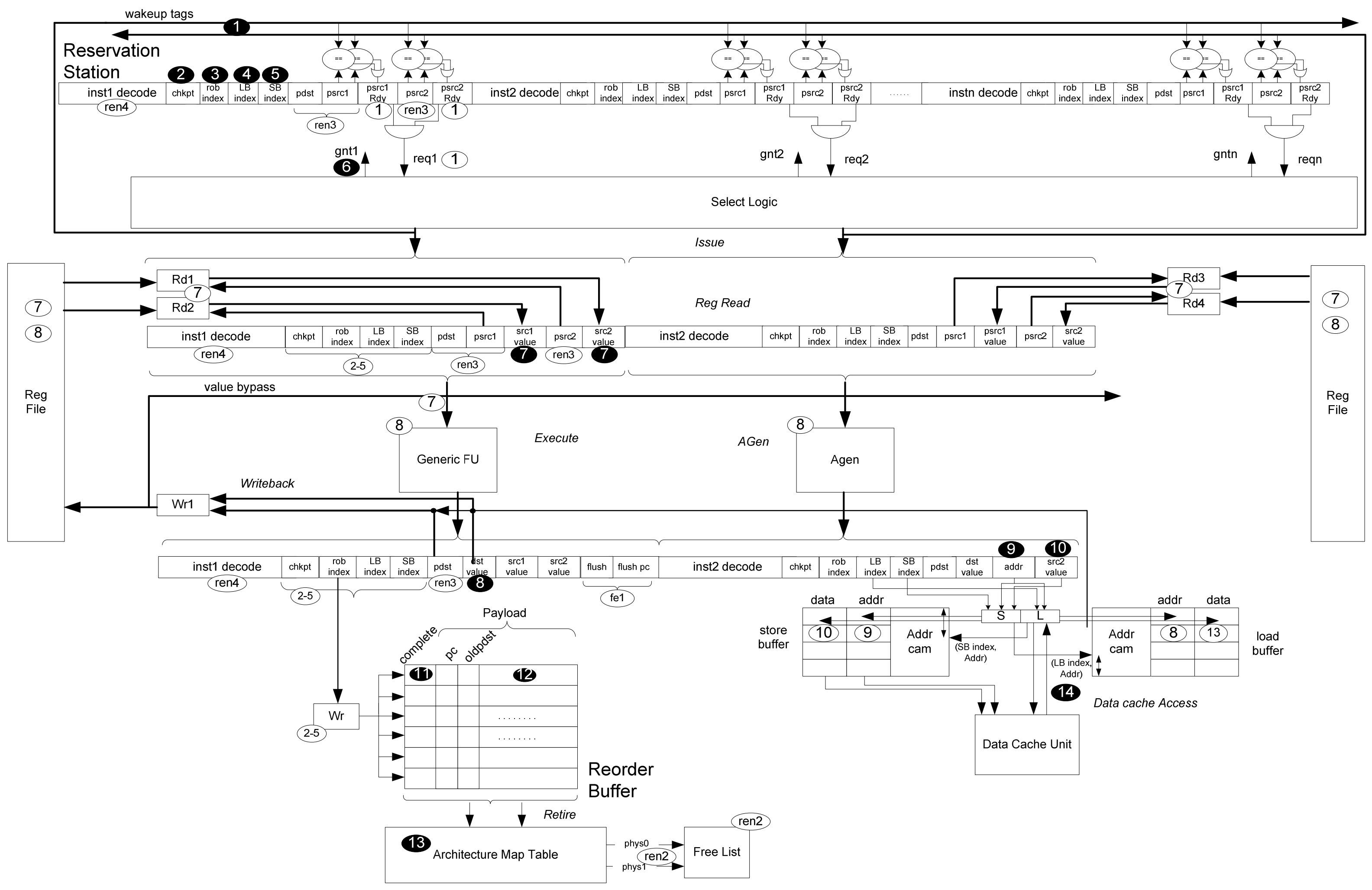


Figure 6-5. Fault analysis of the backend of the pipeline (Issue, Reg Read, Execute/Agen, Writeback, Cache Access, and Retire).

6.3 Fault-Checking Regimen

The fault-checking regimen includes all the fault checks discussed in previous chapters: ITR, RNA, TAC and confident branches as checkers. In addition, new checks are added to the regimen (described below) for higher fault coverage: sequential PC check at retirement to detect faults that cause control-flow deviations and a consumer counter check to detect faults related to source register mappings. Further, the branch pre-decode pipeline stage, already present in most processor pipelines to verify the next PC prediction, inherently masks some transient faults in the fetch unit, and its fault detection capabilities is studied as part of the fault-checking regimen. The full fault-checking regimen is described below.

Inherent time redundancy check (ITR): An ITR cache is included in the pipeline to store instruction trace signatures obtained by combining instruction decode signals, as described in Chapter 3. There is a key change from the pipeline setup described in Chapter 3 : the signature generation logic and the ITR cache are moved from the decode stage to the retirement stage. The pipeline is modified such that instructions also write their decode signals into the ROB when they drain from the pipeline. As instructions retire from the ROB, their decode signals are combined into a signature and the ITR cache is consulted to check for transient faults. Moving the ITR check to the retirement stage extends fault coverage to decode signals across all pipeline stages, in addition to protecting the fetch and decode units. The ITR cache used is a 2-way set-associative cache capable of holding 1024 signatures.

Register Name Authentication previous-mapping check (RNA1) and writeback state check (**RNA2**): The two RNA checks are introduced into the pipeline to detect faults related to destination register renaming, as described in Chapter 4.

Timestamp-based assertion check (TAC): The TAC-Lat check described in Chapter 5 is introduced into the pipeline to detect faults that cause instructions to issue in the wrong order.

Confident branch misprediction check (CONFBR): Branch confidence [77] is tracked and mispredictions among confident branches are viewed as detection of transient faults [30][42][66]. On detecting a confident branch misprediction, the pipeline is flushed and restarted from the instruction at the head of the ROB [42]. The CONFBR check detects some faults affecting values that directly or indirectly feed into a branch, for example, faults on the register file.

Sequential PC check (SPC) at retirement: The sequential PC check was briefly discussed in Section 3.3.5. The idea is to maintain a retirement PC and assert that a committing instruction's PC matches the retirement PC. The retirement PC is updated as follows. Sequential committing instructions add their length (which can be recorded at decode for variable-length ISAs) to the retirement PC and branches update the retirement PC with their calculated PC. Comparing a committing instruction's PC with the retirement PC will detect a discontinuity between two otherwise sequential traces. The SPC check can detect faults that affect sequential control-flow, for example, faults on the PC or a ROB bookkeeping fault that might cause some valid instructions in the ROB to be overwritten.

Register consumer counter (CC) check: The consumer counter check aims to detect faults purely related to source register mappings. As discussed in Section 4.3, such faults cannot be detected by RNA, and only a subset which cause deadlocks are detected by the watchdog timer. The CC check detects such faults by maintaining a counter that indicates the number of consumers for a physical register, and asserting that the consumer count is non-zero when

the register is read. The counter is incremented at the rename stage, when source physical register mappings are read from the rename map table. At the register read stage, it is asserted that the consumer count of a register being read is non-zero, following which, the counter is decremented. Speculative decrements due to misspeculated instructions are undone by scanning the ROB backwards on a misprediction and incrementing the consumer counts of affected source registers.

Branch pre-decode to verify BTB prediction (BTBV): In many processor pipelines, the branch pre-decode stage confirms the correctness of the BTB target address by decoding instructions in a fetched packet and checking for a branch. If a branch is detected and the decoded target address is inconsistent with what the BTB provided, the fetch unit is redirected to the correct PC, and instructions in the interim are flushed. This mechanism has inherent capability to detect transient faults affecting the next PC prediction unit (for example, the BTB target address), and is included as part of the fault-checking regimen.

6.4 Experimental Methodology

The faults identified in Section 6.2 are modeled in a timing simulator. The simulator models microarchitecture similar to the MIPS R10000 processor [41]. The microarchitecture configuration used shown in Table 6-1. A subset of the SPEC2K benchmark suite is used for evaluation. For each benchmark, one thousand faults are randomly injected. Fault injection involves randomly selecting a fault to inject from the list of faults, and triggering the modeled anomaly in the timing simulator. A separate “golden” (fault-free) simulator is run in parallel with the faulty simulator. When an instruction is committed to the architectural state in the faulty simulator, it is compared with its golden counterpart to determine whether or not

the architectural state is being corrupted. Any fault that leads to corruption of architectural state is classified as a potential silent data corruption (SDC) fault. Likewise, if no corruption of architectural state is observed for a set period of time after a fault is injected (the observation window), it is classified as a masked fault. In this study, an observation window of one million cycles is used. A watchdog timer (shown as WDOG in the results) is included in the experiments to check for deadlocks [2][80].

Table 6-1. Microarchitecture configuration.

L1 I & D Caches	64KB, 4-way, 64B line, LRU, L1hit = 1 cycle, L1miss/L2hit = 10 cycles
L2 Unified Cache	1MB, 8-way, 64B line, LRU, L1miss/L2miss = 100 cycles
Branch Predictor	gshare, 16-bit history, 2^{20} entries
Superscalar Core	reorder buffer (ROB): 64 dispatch/issue/retire bandwidth: 4 per cycle

An injected fault leads to one of many possible outcomes, based on the combination of the effect of the fault on the application and, whether or not the fault is detected by the fault-checking regimen. Figure 6-6 shows the possible fault outcomes as a chart. A fault is shown in a black box, its effects on an application in grey boxes, and the final outcomes in white boxes. The possible effects of a fault on an application are, 1) control-flow deviation (CFD), 2) silent data corruption (SDC), 3) application deadlock (Deadlock/WDOG), and 4) masking of the fault (Mask), i.e., it does not have any of the previous effects. If an injected fault is detected, the fault outcome is indicated by prefixing the effect of the fault with the letter A, signifying an assertion. If an injected fault is undetected, the fault outcome is indicated by prefixing the effect of the fault with the letter U. Based on this, the list of possible fault injection outcomes are (also shown in Figure 6-6):

- ACFD: The fault caused a control-flow deviation, and was detected by the fault-checking regimen.

- UCFD: The fault caused a control-flow deviation, and was not detected by the fault-checking regimen.

- ASDC: The fault caused a silent data corruption, and was detected by the fault-checking regimen.

- USDC: The fault caused a silent data corruption, and was not detected by the fault-checking regimen.

- AWDOG: The fault caused a deadlock, and was detected by the fault-checking regimen before the deadlock occurred.

- UWDOG: The fault caused a deadlock, and was not detected by the fault-checking regimen.

- USDCWDOG: The fault caused a silent data corruption and then a deadlock, and was not detected by the fault-checking regimen.

- AMASK: The fault was masked, and was detected by the fault-checking regimen.

- UMASK: The fault was masked, and was not detected by the fault checking regimen.

In a small subset of fault injections pertaining to decode signals, the fault is observed to be undetected at the end of the observation window, but the faulty signal is still present in the ITR cache. Since the fault may get detected by the ITR check at a later point in time, beyond the observation window, the fault outcome is termed unknown (UK). Since unknown faults

are a tiny fraction of fault outcomes, they are not further sub-classified based on their effects, and are shown as one group in the results.

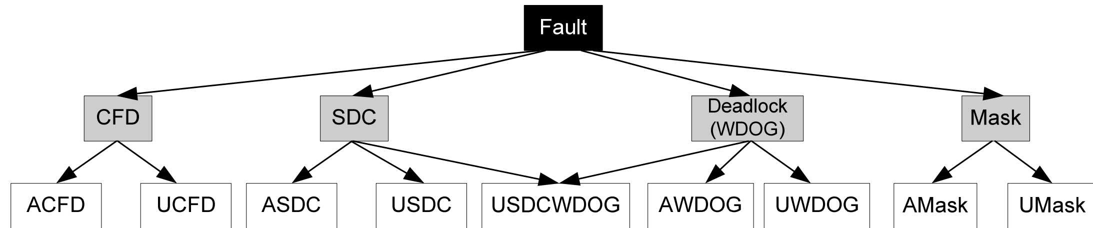


Figure 6-6. Classification chart for fault injection outcomes.

Note that the ordering in time between fault detection and the fault effect is not important in this study, as the focus is on highlighting the fault detection capabilities of the regimen, and the effect on the application if a fault is undetected.

A point worth mentioning is that masking of a fault, and failure to detect a fault, are observations made within the scope of the observation window in the experiments (one million cycles). While this result could change in a longer application run, i.e., a fault that appears to be masked or undetected within the observation window later has an effect on the application or is detected by a check, respectively, the available latency for a fault in the microarchitecture to surface at application level or get detected is usually much smaller than one million cycles, due to pipeline flushes on branch mispredictions, context switches, etc. Hence, the observations made in the experiments are highly likely to be correct, and exception cases like UK faults due to the ITR cache are rare.

6.5 Results

Figure 6-7 shows the distribution of injected faults across all pipeline stages. Faults are distributed in proportion with the number of faults modeled in each stage. A pipeline stage

that models more faults has a larger fraction of faults injected in it. Across benchmarks, the fault distribution pattern is fairly uniform, but not the same due to the randomness in fault injection. On average, the percentage of faults injected in each stage are: fetch – 9%, decode – 39%, rename – 24%, dispatch – 7%, and backend stages – 21%.

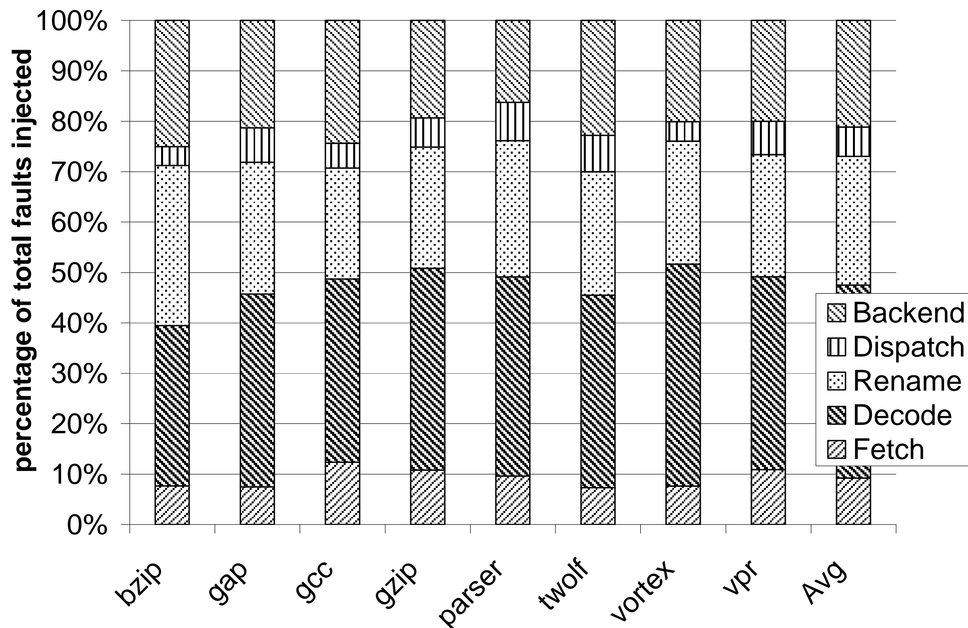


Figure 6-7. Distribution of faults injected across different pipeline stages.

Figure 6-8 shows the fault outcome distribution across all pipeline stages. The distribution pattern across all benchmarks is fairly uniform, and discussion is focused on average results.The average breakdown of faults is as follows: those detected by the regimen is 60%, those not detected by the regimen but detected by the watchdog timer is 9%, and those undetected is 31%.

Several interesting conclusions can be made from the results. Around 40% of faults (= sum of all CFD, SDC, and WDOG causing faults) cause harm to the application being run,

either by corrupting the architectural state, committing wrong instructions or causing a deadlock. This fairly large percentage is good motivation to protect processors from transient faults. Among all faults detected by the fault-checking regimen (ACFD + ASDC + AWDOG + AMASK), the part that causes harmful effects is 24% (ACFD + SDC + AWDOG). So,

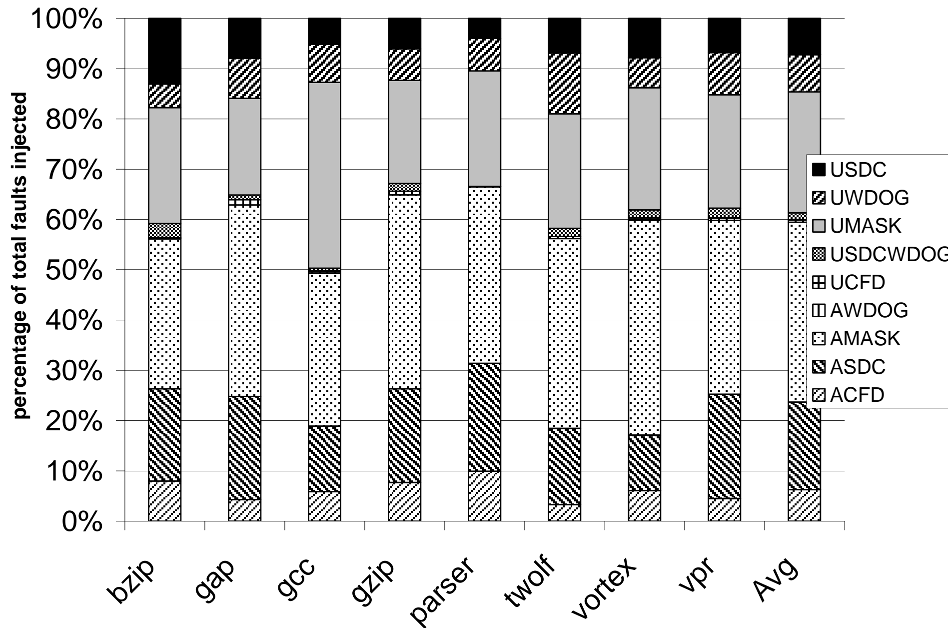


Figure 6-8. Fault outcome distribution.

with the fault-checking regimen in place, the processor is vulnerable to only 40 – 24 = 16% of faults, a 60% reduction in vulnerability. If a watchdog timer is included in the processor, then an additional 9% of faults (that cause deadlocks) can be detected through timeouts (UWDOG + USDCWDOG). This further reduces vulnerability to harmful faults from 16% to 7%. The overall improvement in vulnerability to harmful faults by combining the fault-checking regimen with the watchdog timer is 40% to 7%, an 83% improvement. This is a substantial result for considering a regimen-based approach to processor fault tolerance.

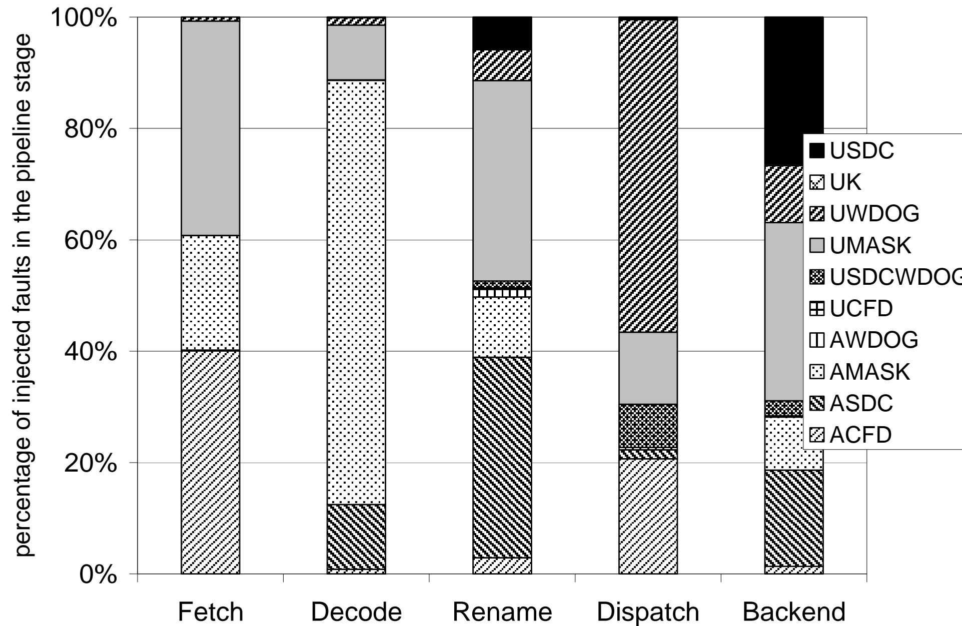


Figure 6-9. Fault outcome distribution per pipeline stage.

To further analyze fault outcomes, the outcome distribution in each pipeline stage is considered, as shown in Figure 6-9. This provides a high-level reference to analyze fault outcomes, showing the pipeline stage(s) where a given fault outcome tends to occur. It will be used as the first step in investigating a fault outcome, followed by looking at a further breakdown of fault outcomes per fault type injected in a pipeline stage, which are shown in Figure 6-10 (fetch), Figure 6-12 (decode), Figure 6-14 (rename), Figure 6-16 (dispatch) and Figure 6-18 (backend). To provide more insight into fault detection, a breakdown of fault checks per fault type in a pipeline stage is shown in Figure 6-11 (fetch), Figure 6-13 (decode), Figure 6-15 (rename), Figure 6-17 (dispatch), and Figure 6-19 (backend).

As seen in Figure 6-9, fault outcomes that cause control-flow deviation (*CFD) mainly occur in fetch and dispatch pipeline stages, with minor incidences in decode, rename and backend stages. For insight, we refer to the outcome breakdown per fault type in the

various pipeline stages. Looking at Figure 6-10, we see that four among the five fault types (FETCH_PC, WRONG_INSTR, INSTR_DISAPP, and FETCHQ) are capable of causing breaks in control flow. As expected, instances of these four fault types are the main contributors to causing a control-flow deviation (see ACFD contribution). All instances of the predicted next PC fault (NEXT_PC) get detected early in the pipeline by the BTBV check, which leads to a subsequent flush and restart of the fetch stage from the correct PC, hence, masking such faults. Looking at Figure 6-16, we find that only the ROB_WRITE fault type contributes to CFD. This is expected, because writing to wrong ROB entries, breaks control flow. Notice from the fault check distributions in Figure 6-11 (for fetch) and Figure 6-17 (for dispatch), no instance of CFD-causing fault is undetected. Moreover, majority of them are detected through the SPC check, and some, through the ITR check.

A small number of CFDs are also caused by faults injected in decode, rename and backend stages. Referring to the respective breakdowns per fault type of these pipeline stages (Figure 6-12 for decode, Figure 6-14 for rename, and Figure 6-18 for issue) reveals that these instances of CFD are caused by faults leading to incorrect branch computation (e.g., SRC_LOG_REG, IMM etc. in decode, REN_MAP_TABLE etc. in rename, and READY_BIT, SPEC_LOAD, etc. in the backend). Most of the faults are aptly detected (see ACFD) by ITR in decode, RNA in rename, and TAC in the backend, and a very small fraction goes undetected (see UCFD).

Next, we explore outcomes that lead to silent data corruption (*SDC). As seen in Figure 6-9, the majority of SDC instances occur in the rename and backend stages. For insight we refer to their respective breakdowns per fault type in Figure 6-14 (for rename) and

Figure 6-18 (for backend). In the rename stage, USDC instances dominantly occur due to the REN_MAP_SRC_INDEX fault type. The anomaly modeled here is a fault while indexing into the rename map table, which is highly likely to cause an instruction to produce a wrong value and cause a SDC. There is no specific fault check in the regimen that is targeted to detect it (the consumer counter check only applies to reading wrong sources after renaming). Partial coverage is provided by TAC, which can detect faults if the fault-afflicted instruction issues before its actual producer, indicated by its non-faulty logical source registers. But a large fraction of the faults are undetected and end up causing an SDC.

Referring to Figure 6-18, USDC instances often occur due to faults that directly or indirectly affect an output value (e.g., SRC_VALUE, DST_VALUE, etc.). Such faults are very likely to corrupt an output value that can influence the architectural state, hence, causing SDC. The regimen does not provide comprehensive coverage to values through any of the fault checks. Partial coverage is provided by misprediction detection among confident branches. But the number of confident branches are limited, and hence, a large number of such faults go undetected (e.g., all SRC_VALUE faults are undetected) and end up causing SDC.

The other major contributors to USDC are faults in the rename structures, as seen in Figure 6-14. Though faults in these structures get comprehensive coverage from RNA, some intervening system calls do not provide the opportunity to detect these faults using RNA. Similar results were seen in the experimental evaluation of RNA in Chapter 4 (see Section 4.4.2).

Next, fault outcomes that lead to deadlocks are analyzed (*WDOG). We observe from Figure 6-9, that almost all deadlocks are detected by the watchdog timer (UWDOG and USDCWDOG), and only a small fraction is detected first by regimen-based checks (AWDOG). This underscores the advantage of including a watchdog timer in a processor. From Figure 6-9, most of the deadlocks are caused by faults in the dispatch stage. For insight, we look at the per fault type breakdown for the dispatch stage in Figure 6-16. As seen there, deadlocks are caused by all the fault types. This is expected. The ROB_WRITE and RS_WRITE faults result in writing instructions to wrong locations in the ROB and reservation stations, respectively. If instructions wrongly overwrite other instructions their dependents end up waiting endlessly for results to become available, and hence cause a deadlock. The LSQ_WRITE fault results in a load or store being written to a wrong entry in the load or store buffer. If an unexecuted load or store is overwritten, then the .complete status of that load or store is never updated in the ROB, resulting in a deadlock. The other major contributors to outcomes causing deadlocks are faults in the backend, mainly related to the wakeup mechanism (READY_BIT, SPEC_LOAD, WAKEUP_TAG, etc. as shown in Figure 6-18). They are also detected by the watchdog timer.

Finally, we explore fault outcomes that are masked. A large fraction of faults are masked – 60% of the faults from Figure 6-8. From Figure 6-9, we find that all pipeline stages have substantial number of outcomes that are masked. There are two main reasons why injected faults lead to masked outcomes: 1) speculative microarchitecture state due to branch mispredictions, and 2) injecting faults in locations that are dormant, i.e., microarchitecturally dead. To find out which was a more significant factor for masking, a deeper analysis was

performed by simulating perfect branch prediction (not shown in the results). The goal was to rule out masking due to misspeculation by avoiding branch mispredictions in the experiments. The analysis revealed that, with perfect branch prediction, masking was reduced only by a small margin, hence, concluding that most of the masking is due to injecting faults in places that are not required for architecturally correct execution [10]. For example, many of the faults in the decode stage were injected on a decode signal that was not relevant to the associated instruction, leading to masking of the fault.

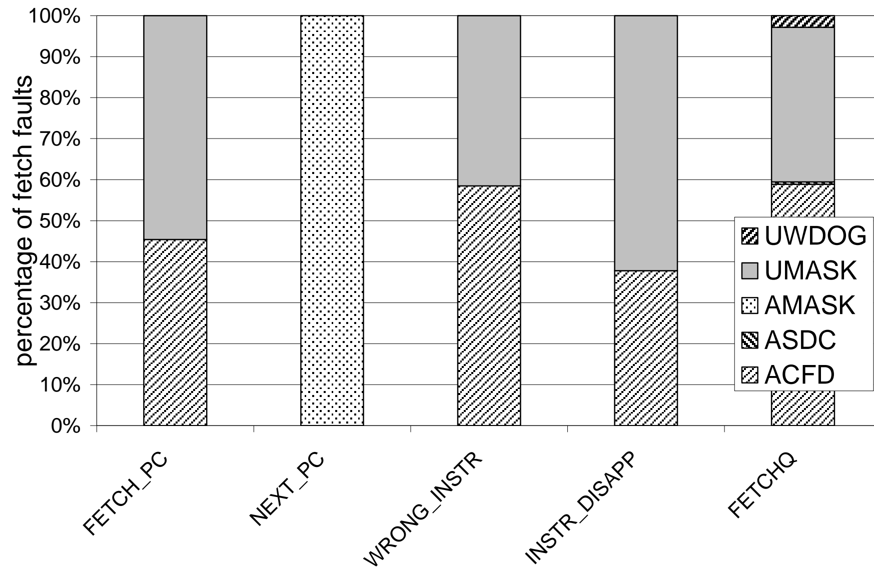


Figure 6-10. Fault outcome distribution for each fault type injected in the fetch stage.

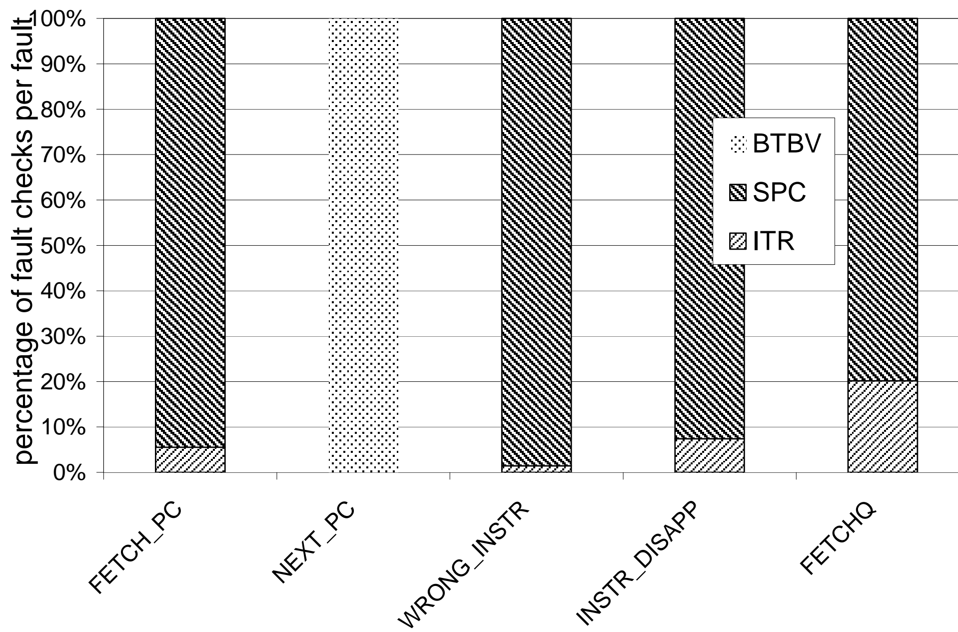


Figure 6-11. Fault check distribution for fetch stage.

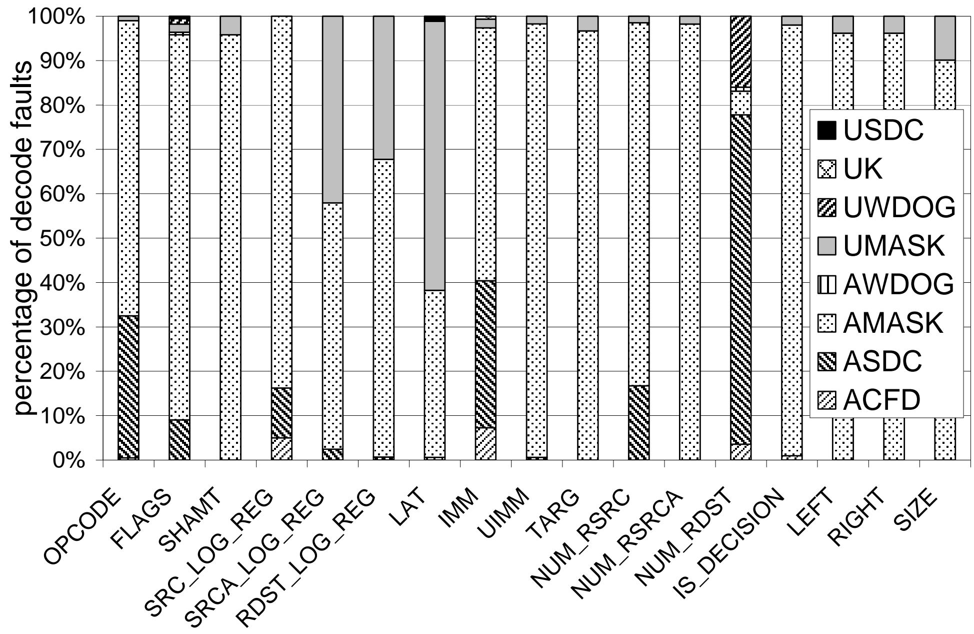


Figure 6-12. Fault outcome distribution for each fault type injected in the decode stage.

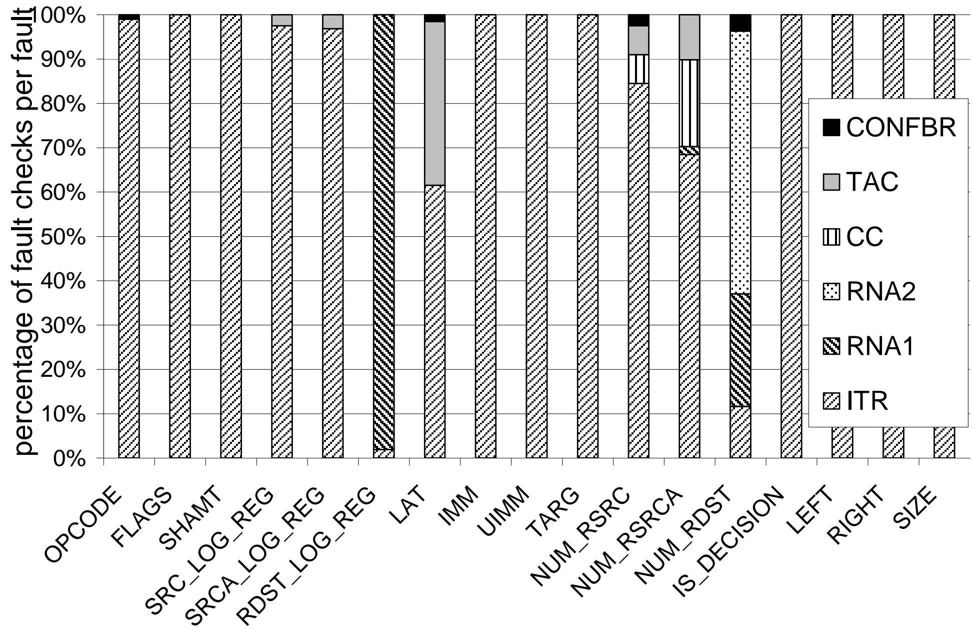


Figure 6-13. Fault check distribution for decode stage.

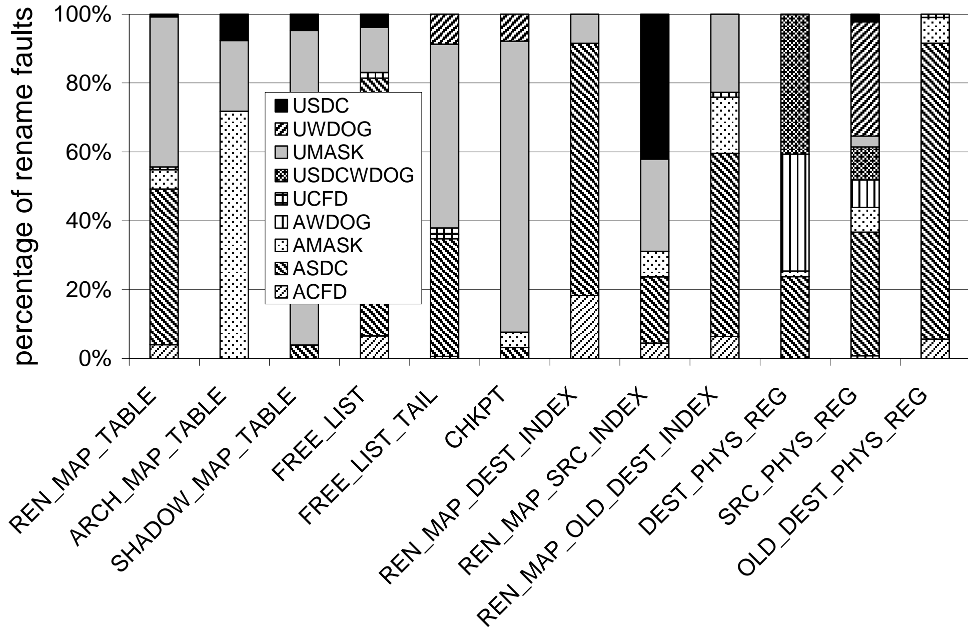


Figure 6-14. Fault outcome distribution for each fault type injected in the rename stage.

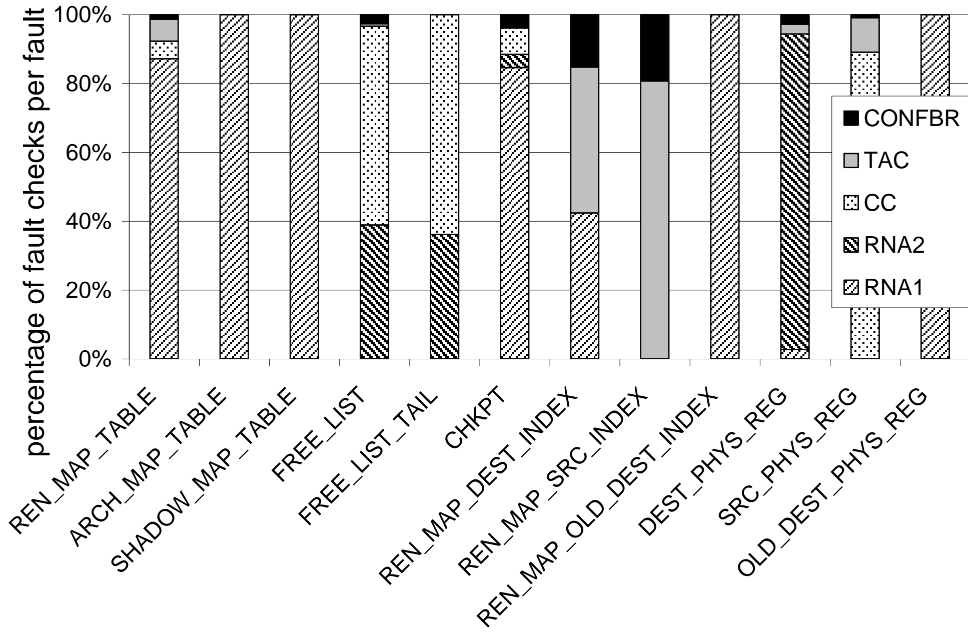


Figure 6-15. Fault check distribution for rename stage.

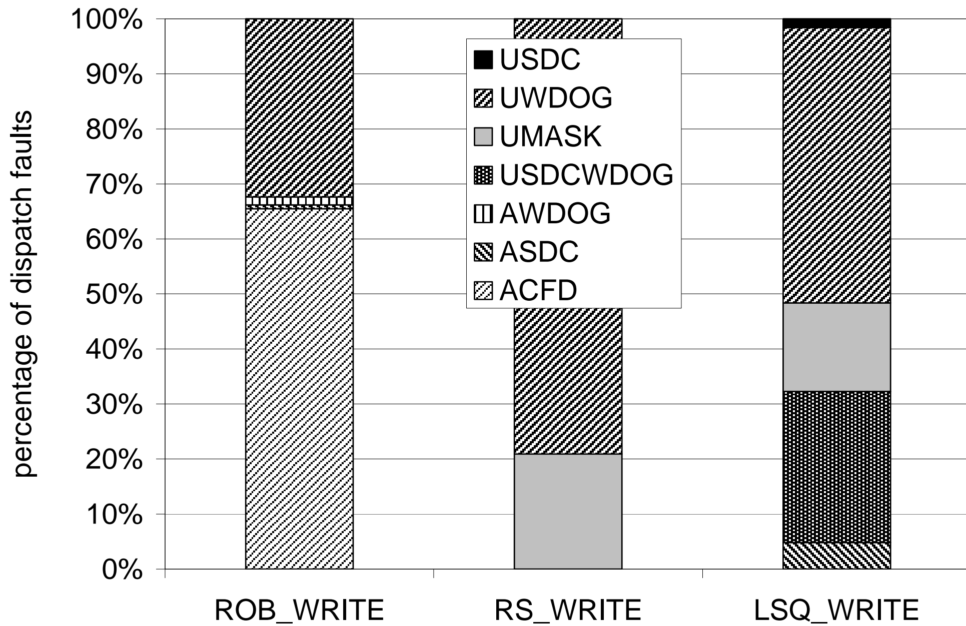


Figure 6-16. Fault outcome distribution for each fault type injected in the dispatch stage.

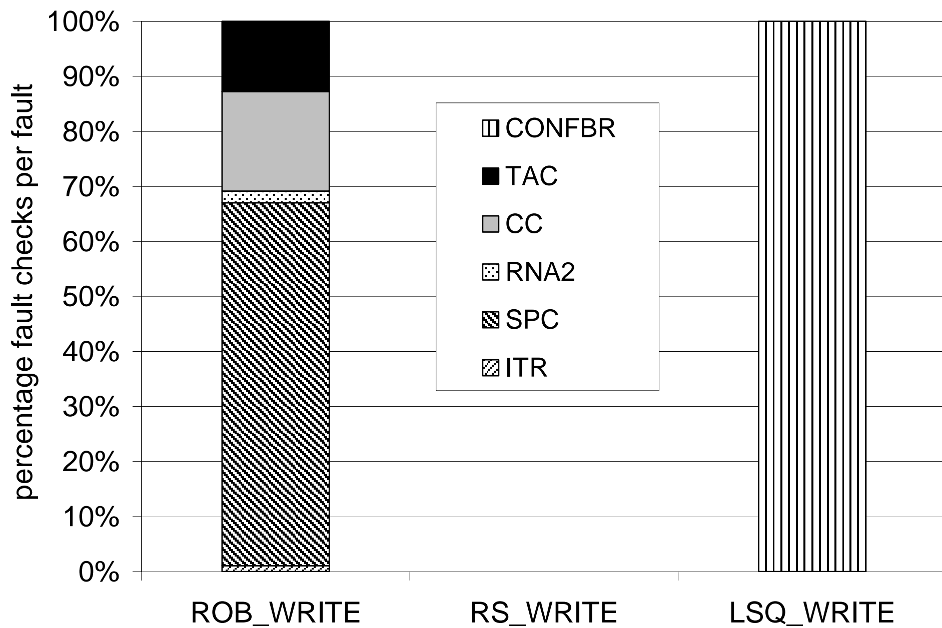


Figure 6-17. Fault check distribution for dispatch stage.

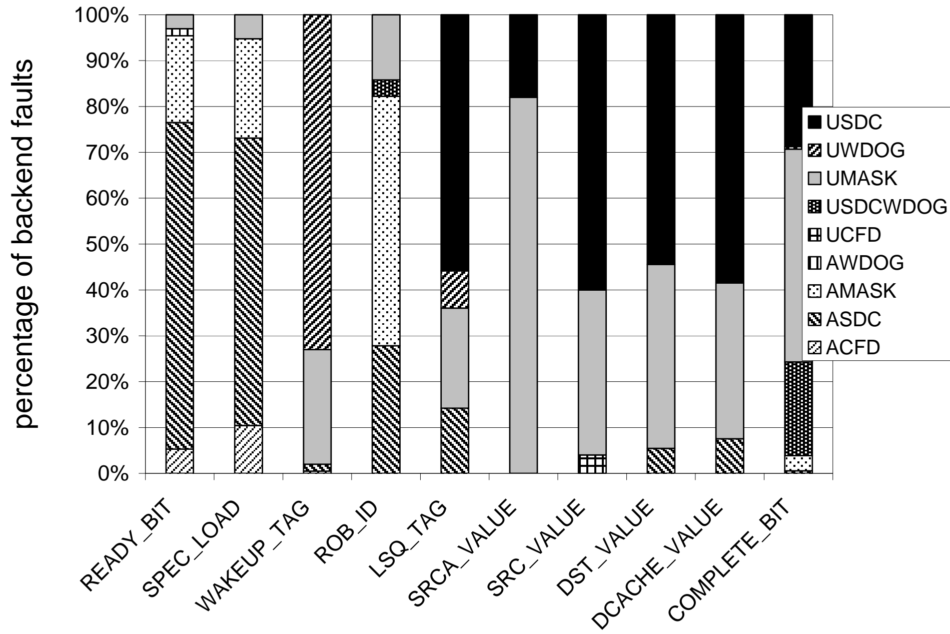


Figure 6-18. Fault outcome distribution for each fault type injected in the backend stages.

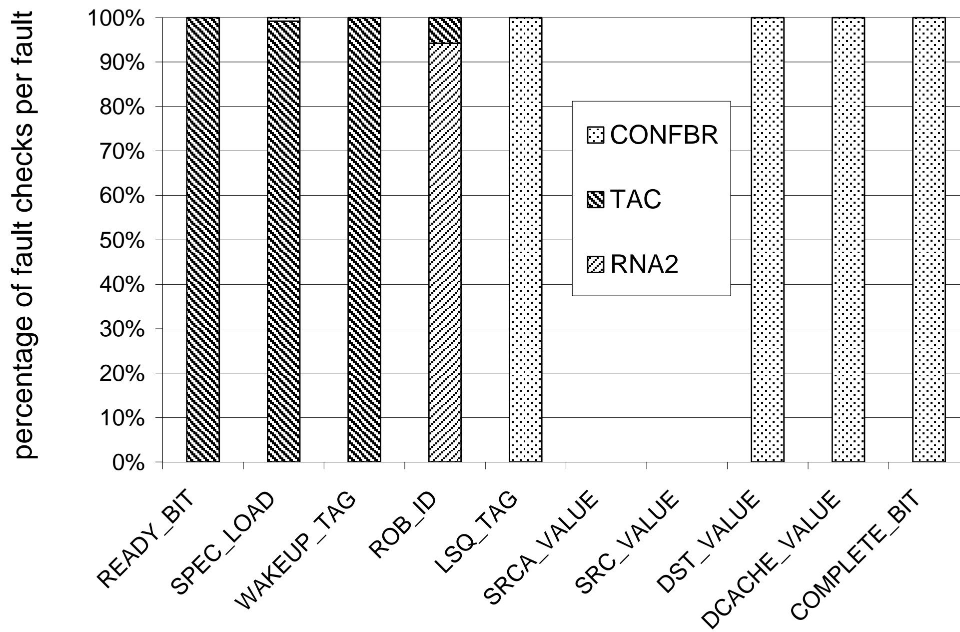


Figure 6-19. Fault check distribution for backend stages.

Chapter 7: Related Work

This chapter discusses prior work on processor transient fault tolerance. Previously proposed transient fault tolerance approaches can be broadly classified into four categories: space redundancy based approaches, time redundancy based approaches, information redundancy based approaches and logic-circuit level approaches.

7.1 Space Redundancy Based Approaches

The general theme in space redundancy is to replicate components and compare outputs of the replicated components to detect transient faults. A classic example of space redundancy is the IBM S/390 mainframe processor [51][52], where the I-unit (fetch and decode units) and E-unit (execution unit) are duplicated, and their signals compared for transient fault detection.

In N-modular processor redundancy, N processors perform the same task and compare results at an external chip interface to detect transient faults [62]. A voting mechanism compares the outputs and chooses the final output on a majority basis (if N is greater than two). Fault-tolerant computer systems using Triple Modular Redundancy (TMR) are a popular example of N-modular redundancy where three processors perform the same task in lockstepped execution and compare results at an external chip interface [62][63]. A voter compares the three results and chooses the final result on a majority basis. TMR has been the prominent fault tolerance solution in aircrafts (e.g., [64]) and space shuttles, where not only

processors, but entire systems are replicated for robustness. TMR incurs high area and power overheads, due to the replicated processors and the voting circuitry.

Dual Modular Redundancy (DMR) is another example of N-modular redundancy where two replicas of a module execute in lockstepped fashion and compare their outputs to detect transient faults. A classic example of DMR is the Nonstop server developed by Tandem [40], where two processors run in lockstepped fashion and compare their results to detect transient faults.

Lockstepped execution has significant implementation challenges, especially for modern chip multiprocessors (CMP). For example, in modern CMPs, processors may have separate clock domains for dynamic frequency control purposes, making lockstepped execution impractical [65]. There are other potential showstoppers for implementing lockstep in modern CMPs [65][35]. There have been research proposals for CMP-based DMR that do not use lockstepped execution. The general idea is to bind the two processors architecturally rather than synchronizing their timing in hardware. Slipstream on a CMP [16][15][27] uses a staggered execution model where one processor executes ahead of the second processor. The two processors are synchronized by a delay buffer comprising of results from the leading processor. The trailing processor consumes these results as predictions, and flags transient faults in the form of mispredictions.

Chip-level Redundant Threading (CRT) [9][4] uses an input replication and output comparison approach, originally proposed for SMT processors [20], for synchronizing the two processors. Rather than checking the outcomes of all instructions, only memory store addresses and values are checked, and to guarantee that both processors get the same inputs,

memory load values are replicated in both the processors. A store queue is used for synchronizing and comparing stores, and a load value queue (LVQ) or an active load address buffer (ALAB) is used to replicate values loaded by the leading processor in the trailing processor [20]. Transient faults surface as mismatches in store addresses or values. There has also been recent work that shows input replication in CRT can be relaxed, and input incoherence can be corrected by leveraging existing mechanisms used for detecting and recovering from soft errors [65]. Input incoherence and correction is leveraged in Slipstream to permit simple and efficient memory renaming [17].

Another research proposal that optimizes DMR-based execution is Fingerprinting [78]. A cryptographic hash value (called a fingerprint) is generated for updates to architectural state that spans several instructions, and checking take place by comparing fingerprints of two processors at an external chip interface. Fingerprinting significantly reduces the I/O bandwidth required for comparing processor outputs.

In another direction, there have also been proposals to exploit N-modular redundancy at a software level. In one approach [86], two virtual machines are run on separate processors and their outputs compared in software to check for transient faults. The study develops mechanisms to guarantee coherency in the two instruction streams. Basically, instructions that would cause non-deterministic changes to the virtual machine state are identified, and their effects replicated identically in the two virtual machines. This includes coordinating I/O operations [86]. In another approach [91], operating system processes are triplicated and run on multiple available cores. Input replication and output comparison is done by a system call emulation unit. Basically, at each system call, the system call emulation unit is invoked,

which performs input replication and output comparison. In the case of fault detection, the emulation unit performs voting to determine the correct output.

7.2 Time Redundancy Based Approaches

Time redundancy is achieved by executing a program redundantly at different times on the same hardware.

In Redundant Execution with Shifted Operands (RESO) [39][60], some instructions are executed redundantly with shifted operands on the same functional units. Shifting the result back by the same amount yields the original result computed with unshifted operands. While re-executing instructions detects transient faults, re-executing them with shifted operands also detects permanent faults.

In another direction, there have been proposals to introduce redundant execution by modifying conventional superscalar pipelines [19][37][26][59].

A more comprehensive time redundant execution model is redundant multithreading. In redundant multithreading architectures [20][23][27][29], all instructions are fetched and executed twice, via two redundant threads on a simultaneous multithreading (SMT) pipeline. The results of the duplicated instructions are compared to detect transient faults in the entire pipeline. There has been significant interest in redundant multithreading architectures in recent years. The first redundant multithreading proposal was AR-SMT [23]. One of the key contributions of AR-SMT was binding the two threads architecturally using a staggered execution model as opposed to lockstep execution used in previous architectures (e.g., [40]), where the two threads were synchronized by matching their timing in hardware. Lockstepping has implementation challenges [65] and, moreover, can degrade performance

[9]. In AR-SMT, the leading thread is staggered from the trailing thread by a short distance equal to the size of a delay buffer. The delay buffer contains results of the leading thread, which are consumed as predictions by the trailing thread. Confirming the correctness of the predictions also checks for transient faults in the pipeline. Several redundant execution based architectures that followed incorporated staggered execution [20][27][4][5][9].

The performance overhead of dual redundant threads on an SMT pipeline can be significant due to resource contention (fetch, issue, and retire bandwidth, physical registers, etc.) and checking bandwidth. A universal goal for all redundant threading architectures has been maximizing both coverage and performance. Thus, we focus discussion of related work on optimizations towards this goal, including (1) reducing resource pressure, (2) reducing checking bandwidth, and (3) reducing instruction count.

Several techniques have been developed to reduce resource contention. First, the leader/follower arrangement of the staggered execution model used in AR-SMT enables a key performance optimization: the leading thread's outcomes can be leveraged as likely-correct (they are correct in the fault-free case) branch and value predictions in the trailing thread [23]. The trailing thread executes more efficiently because all of its control and data dependences are eliminated (no wrong-path instructions and perfect value prediction in fault-free case). The effect is that the trailing thread requires fewer resources than the leading thread for the same performance, thus releasing resources back to the leading thread, reducing overall execution time for dual-redundant execution. This resource optimization has also been used in other redundant execution based architectures [20][27][2]. Microarchitecture-based Introspection (MBI) [73] is another technique that reduces resource

contention by postponing redundant execution of the instruction stream to periods of long-latency cache misses. MBI is well suited for memory-intensive applications.

Recent work proposes other per-structure optimizations for reducing resource pressure [7], such as packing dual instances of a dynamic instruction into the same physical register for short-width values (exploiting advance knowledge from the leading thread) or intelligently reallocating some leading thread's physical registers to the trailing thread to yield a net reduction in physical register pressure.

In another direction, several techniques have been proposed for reducing the number of checks (comparisons). All instructions are still executed twice, so that whichever two instructions are compared, they are still based on separate computation. One approach can detect all single transient faults by checking only store instructions [20]. Dependence Based Checking Elision [29][4] reduces the number of checks based on the idea that a fault propagates through dependent instructions, so checking an instruction in a chain implicitly checks instructions leading to it. Prediction-based PRT approaches [42][66] exploit the same principle to recover from faults on singly executed instructions. In this case, there is no choice but to check only the consumer whereas in DBCE all instructions are redundantly executed with the option of not checking all of them. The key difference is that prediction-based PRT fully capitalizes on the notion of consumer-based checking, by not only eliminating the check of the producer, but the producer instruction itself. This reduces pressure not only on the checking machinery, but the processor as a whole.

A number of varied approaches reduce the number of redundantly executed instructions. Two of these, Slipstream [6][15][16][17] [24][27] and ReStore [30], employ

forms of predictive checking. As mentioned in Chapter 2, past characterization of slipstream fault tolerance is not extensive and yields a coverage bound limited to only redundantly executed instructions. This dissertation contributes new analysis that reveals theoretical coverage of singly executed instructions, and recovery techniques to achieve the coverage. ReStore [30] is closely related in two respects. First, fault detection is achieved purely by symptoms such as exceptions, cache misses, TLB misses, and branch mispredictions, a form of predictive checking since no redundant execution is used at all (although frequent and distant rollbacks are in some sense redundant execution after the fact). Second, when such symptoms are detected, the processor rolls back to a prior distant checkpoint, in the hope that the faulty instruction has not retired. This is similar in spirit to recovery of singly executed instructions in Chapter 2 of this dissertation. ReStore does not exploit redundant execution *a priori* and thus is one extreme of the predictive checking spectrum. The "partial thread" has no computation at all and this has performance implications (frequent rollbacks if any misprediction is a symptom) and/or coverage drawbacks (using only confident branch mispredictions as symptoms limits coverage to their slices only). There has also been recent work that leverages anomalies during speculative execution to detect transient faults [76], in similar spirit as ReStore [30].

Opportunistic fault tolerance [5] also reduces the number of redundantly executed instructions by initiating redundancy only during phases of otherwise poor performance. Singly executed instructions are not covered. Compiler-based approaches [84][85] [21][22][67][68][69][70][71] provide a level of control over performance and coverage not feasible in purely hardware threading approaches, such as complex analytical frameworks for

identifying code regions where performance and coverage are not conflicting. Finally, instruction reuse can be used to reduce the number of redundant executions [12].

A related MS thesis laid the initial groundwork for analyzing prediction-based PRT [14]. It describes a hardware implementation of the forward slice checking analysis. The hardware mechanism explicitly classifies instructions as checked or not checked at retirement, and stalls commit, accordingly. Chapter 2 shows that check status does not need to be literally tracked in hardware and the proposed recovery models achieve similar high coverage without explicit tracking. Chapter 2 also presents an in-depth study of prediction-based partial redundant threading.

Diva [2] is an architecture that uses a simple processor core (the "checker") to check the correctness of a complex processor core. The checker re-executes the retired instruction stream and confirms the results produced by the complex core. The checker core can detect design bugs and transient faults in the complex core. Though the checker is a simple core, it keeps up with the complex core by using retired results to break data dependencies, much like in AR-SMT [23].

7.3 Information Redundancy Based Approaches

Information redundancy based approaches are typically used to protect storage elements (like memory, caches, register files, etc.) using error detecting and/or correcting codes [79]. Additional circuitry is required for code generation when data is written, and for error detection when data is read. Since the additional circuitry can impact processor cycle time, this approach has only been used in permitting processor structures. This excludes

pipeline latches, as they are usually on processor critical paths. Nevertheless, parity and ECC-based protection have been the most widely used protection mechanisms in computers.

Hard disk failures in computers are covered using RAID [93] that employs multiple redundant disk drives to mask disk failures. RAID proposes various levels of data protection that combine data mirroring, ECC, parity, etc.

Parity and ECC have been used to protect main memories from errors caused by natural radiation interference [94]. Early on parity used to be sufficient to detect errors in memories, but as memory errors occurred frequently and led to system crashes, ECC-based protection that could detect and correct single-bit errors, plus detect double-bit errors (known as single error correct, double error detect, or SEC-DED), became a commonplace. The IBM chipkill technology (also referred to as redundant array of inexpensive DRAMs for Memory, or RAID-M) further enhanced ECC-protected memories to be able to recover from entire memory chip failures, by spreading bits of an ECC word across multiple memory chips [95][96].

Parity and ECC are also used to protect cache-related structures in processors. To keep up with high processor speeds, caches are designed with aggressively scaled down nodes, making them particularly vulnerable to soft errors from natural radiation interference, as is evident from critical crashes in commercial servers due to unprotected caches [50]. In the Intel Itanium processors, the L1 instruction and data caches are parity protected, backed-up by ECC-protected L2 and L3 cache arrays [80][82]. The Itanium features a multilevel error containment strategy along with a rigorous machine check abort architecture for error logging and diagnosis [80][92]. In one implementation of the Itanium Processor Family, the

Montecito [82], the integer and floating-point register files are parity protected for soft-error detection [81]. In the AMD Hammer family of processors [97] (e.g., the AMD Opteron [98]), the L1 D-cache array is ECC protected and so are the unified L2 cache array and tags. The I-cache and all TLBs are parity protected. Additionally, a hardware scrubber autonomously reads and writes back L1- and L2-cache lines, correcting any errors in the process [98][99], and a machine check architecture logs errors for diagnosis [98].

7.4 Logic-Circuit-Level Approaches

There have been many proposals to detect and correct transient faults at the logic and circuit levels. This section reviews a subset of them which have been proposed to protect latches at a circuit-level.

Razor [100][101][102] is a circuit-level correction technique that uses a shadow latch besides a normal latch. The shadow latch receives a delayed clock signal, allowing it to capture delayed combinational logic outputs that the normal latched missed in its setup window. Comparing the shadow latch and the normal latch detects circuit-level timing errors. This mechanism can also detect soft errors [102].

Built-in Soft Error Resilience (BISER) [104] is a circuit-level technique that reuses on-chip scan resources for soft error detection. The idea is that scan flip-flops are included in the chip for design-for-test purposes, but are not used in normal operation. By modifying the scan flip-flops to act as shadow flip-flops, they can be leveraged to detect soft errors.

Chapter 8: Summary

This thesis proposed using microarchitecture insights for achieving efficient fault tolerance. New microarchitecture insights were developed, and two low-overhead approaches to fault tolerance were presented based on them.

The first architecture is based on the insight that a correct prediction of an instruction's outcome is the same as the outcome produced by fault-free execution of the instruction. By leveraging this idea, a new prediction-based partial redundant threading (PRT) paradigm was presented as a low-overhead alternative to full redundant multithreading (RMT). In RMT, two copies of a program are executed on a simultaneous multithreading (SMT) substrate. Outcomes of duplicated instructions are compared to detect transient faults in the processor. RMT incurs high performance and power overheads due to full redundant execution. In prediction-based PRT, confident predictions are leveraged as effective proxies for redundant execution, and hence, confidently-predicted instructions and their producers are skipped in the redundant thread. Slipstream was used as a platform to study prediction-based PRT, and it was shown for the first time that Slipstream has high fault detection coverage of instructions (99.9%), as opposed to lower estimates from previous studies that only claimed coverage for duplicated instructions. It was shown that Slipstream's existing recovery implementation fails to capitalize on the excellent fault detection coverage. A suite of microarchitecture alterations were then proposed to enhance Slipstream's fault recovery coverage. The end result of applying prediction-based PRT principles on Slipstream was an

improvised Slipstream processor with close-to single-thread performance and close-to 100% fault coverage.

Next, a superscalar processor was designed with built-in checks to indirectly detect low-level transient faults, by observing microarchitecture-level anomalies they cause. Fault checks at the microarchitecture level have a unique advantage over other solutions in lowering the overheads of fault tolerance. Since they establish correctness of the microarchitecture, they provide broad fault coverage to many logic blocks with a few checks. This dissertation developed several novel microarchitecture-level fault checks. Most notably, 1) inherent time redundancy (ITR) exploits program repetition to detect faults in decode signals, thereby covering the fetch and decode units, 2) register name authentication (RNA) asserts consistencies among renaming structures to detect faults affecting renaming, and 3) timestamp-based assertion checking (TAC) asserts sequential order among dependent instructions to detect faults affecting dynamic instruction scheduling. Using these checks, a fault-checking regimen was engaged to comprehensively protect a superscalar processor pipeline. To evaluate fault tolerance of the processor, a new fault injection strategy was developed. It involved analyzing the microarchitecture of a superscalar processor in depth and identifying high-level faults which can be modeled in a timing simulator, enabling a fast and reasonably accurate evaluation. Fault injection experiments revealed that the fault-checking regimen provides substantial coverage and lowered the processor's vulnerability to harmful (unmasked) faults by 83% on average.

Bibliography

[1] A. Avizienis, J. Laprie and B. Randell. Fundamental concepts of dependability. Research Report N01145, Laboratory for Analysis and Architecture of Systems, Centre National de la Recherche Scientifique (LAAS-CNRS), April 2001.

[2] T. M. Austin. Diva: a reliable substrate for deep submicron microarchitecture design. *32nd International symposium on Microarchitecture (MICRO-32)*, pp. 196-207, 1999.

[3] D. Burger, T. M. Austin, and S. Bennett. The Simplescalar Toolset, Version 2. Tech. Report CS-TR-1997-1342, CS Department, University of Wisconsin-Madison, July 1997.

[4] M. Gomaa, C. Scarbrough, T. N. Vijaykumar, and I. Pomeranz. Transient-fault recovery for chip multiprocessors. *30th Int'l Symposium on Computer architecture (ISCA-30)*, pp. 98-109, 2003.

[5] M. Gomaa and T. N. Vijaykumar. Opportunistic transient-fault detection. *32nd International Symposium on Computer Architecture (ISCA-32)*, pp. 172-183, June 2005.

[6] J. J. Koppanalil and E. Rotenberg. A simple mechanism for detecting ineffectual instructions in Slipstream processors. *IEEE Trans. Comput.*, volume 53, issue 4, pages 399-413, 2004.

[7] S. Kumar and A. Aggarwal. Optimal resource allocation for concurrent error detection techniques in high performance microprocessors. *12th International Symposium on High-Performance Computer Architecture (HPCA-12)*, pp. 212-221, February 2006.

[8] K. M. Lepak and M. H. Lipasti. On the value locality of store instructions. *27th International Symposium on Computer Architecture (ISCA-27),* pp. 182-191, June 2000.

[9] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt. Detailed design and evaluation of redundant multithreading alternatives. *29th International Symposium on Computer Architecture (ISCA-29)*, pp. 99-110, 2002.

[10] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. *36th International Symposium on Microarchitecture (MICRO-36)*, pp. 29-40, 2003.

[11] A. Biswas, P. Racunas, R. Cheveresan, J. Emer, S. S. Mukherjee, and R. Rangan. Computing architectural vulnerability factors for address-based structures. *32nd International Symposium on Computer Architecture (ISCA-32)*, pp. 532-543, 2005.

[12] A. Parashar, S. Gurumurthi and A. Sivasubramaniam. A complexity-effective approach to ALU bandwidth enhancement for instruction-level temporal redundancy. *31st International Symposium on Computer Architecture (ISCA-31)*, pp. 376-386, 2004.

[13] J. J. Koppanalil. A Simple Mechanism for Detecting Ineffectual Instructions in Slipstream Processors. *M. S. Thesis, ECE Department, North Carolina State University,* May 2002.

[14] S. Parthasarathy. Improving transient fault tolerance of Slipstream processors. *M.S. Thesis, ECE Department, North Carolina State University,* December 2005.

[15] Z. R. Purser. Slipstream processors. *Ph.D. Thesis, ECE Department, North Carolina State University,* July 2003.

[16] Z. Purser, K. Sundaramoorthy, and E. Rotenberg. A study of Slipstream processors. *33rd International Symposium on Microarchitecture (MICRO-33)*, pp. 269-280, 2000.

[17] Z. Purser, K. Sundaramoorthy, and E. Rotenberg. Slipstream Memory Hierarchies. ECE Tech. Report, North Carolina State University, 2002.

[18] N. Gupta. Slipstream-Based Steering for Clustered Microarchitectures. *M.S. Thesis, ECE Department, North Carolina State University,* August 2003.

[19] J. Ray, J. C. Hoe and B. Falsafi. Dual use of superscalar datapath for transient-fault detection and recovery. *34th International Symposium on Microarchitecture (MICRO-34)*, pp. 214-224, 2001.

[20] S. K. Reinhardt and S. S. Mukherjee. Transient fault detection via simultaneous multithreading. *27th International Symposium on Computer architecture (ISCA-27)*, pp. 25-36, 2000.

[21] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan and D. I. August. SWIFT: Software implemented fault tolerance. *3rd International Symposium on Code Generation and Optimization (CGO-3)*, pp. 243-254, March 2005.

[22] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August. Design and Evaluation of Hybrid Fault-Detection Systems. *32nd International Symposium on Computer Architecture (ISCA-32)*, pp. 148-159, 2005.

[23] E. Rotenberg. AR-SMT: A microarchitectural approach to fault tolerance in microprocessors. *29th International Symposium on Fault-Tolerant Computing (FTCS-29)*, p. 84, 1999.

[24] E. Rotenberg. Exploiting large ineffectual instruction sequences. *Technical Report, North Carolina State University,* November 1999.

[25] J. E. Smith and A. R. Pleszkun. Implementation of Precise Interrupts in Pipelined Processors. *12th International Symposium on Computer Architecture (ISCA-12)*, pp. 36-44, June 1985.

[26] J. C. Smolens, J. Kim, J. C. Hoe and B. Falsafi. Efficient Resource Sharing in Concurrent Error Detecting Superscalar Microarchitectures. 37th *International symposium on Microarchitecture (MICRO-37)*, pp. 257-268, 2004.

[27] K. Sundaramoorthy, Z. Purser, and E. Rotenberg. Slipstream processors: improving both performance and fault tolerance. *9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-9)*, pp. 257-268, 2000.

[28] D. Tullsen, S. J. Eggers and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. *22nd International Symposium on Computer Architecture (ISCA-22)*, 1995.

[29] T. N. Vijaykumar, I. Pomeranz, and K. Cheng. Transient-fault recovery using simultaneous multithreading. *29th International Symposium on Computer architecture (ISCA-29)*, pp. 87-98, 2002.

[30] N. J. Wang and S. J. Patel. ReStore: Symptom based soft error detection in microprocessors. *35th International Conference on Dependable Systems and Networks (DSN-35)*, pp. 30-39, 2005.

[31] Z. Kalbarczyk, R.K. Iyer, G.L. Ries, J.U. Patel, M.S. Lee, and Y. Xiao. Hierarchical simulation approach to accurate fault modeling for system dependability evaluation. *IEEE Transactions on Software Engineering,* volume 25, issue 5, pages: 619-632, September/October 1999.

[32] G. Saggese, N. J. Wang, Z. Kalbarczyk, S. J. Patel, and R. K. Iyer. An Experimental Study of Soft Errors in Microprocessors. *IEEE Micro*., Volume 25, Issue 6, pp: 30-39, November 2005.

[33] P. Liden, P. Dahlgren, R. Johansson, and J. Karlsson. On latching probability of particle induced transients incombinational networks. *24th International Symposium on Fault-Tolerant Computing (FTCS-24)*, pp. 340-349, 1994.

[34] P. Shivakumar, M. Kistler, S. W. Keckler, D. Burger, and L. Alvisi. Modeling the Effect of Technology Trends on the Soft Error Rate of Combinational Logic. *32nd International Conference on Dependable Systems and Networks (DSN-32),* pp: 389-398, 2002.

[35] S. Borkar, T. Karnik, S. Narendra, J. Tschanz, A. Keshavarzi, and V. De.. Parameter variations and impact on circuits and microarchitecture. *40th Design Automation Conference,* pp: 338-342, 2003.

[36] N. J. Wang, J. Quek, T. Rafacz, and S. J. Patel. Characterizing the effects of transient faults on a high-performance processor pipeline. 34th *International Conference on Dependable Systems and Networks (DSN-34),* pp: 61-70, 2004.

[37] S. Kim and A. K. Somani. SSD: An affordable fault tolerant architecture for superscalar processors. *8th Pacific Rim International Symposium on Dependable Computing (PRDC-8), pp:* 27-34, 2001.

[38] M. Nicolaidis. Efficient implementations of self-checking adders and ALUs. *23rd International Symposium on Fault-Tolerant Computing (FTCS-23)*, pp: 586-595, June 1993.

[39] J.H. Patel and L. Y. Fung. Concurrent error detection in ALUs by recomputing with shifted operands. *IEEE Transactions on Computers*, pages 589-595, July 1982.

[40] D. McEvoy. The Architecture of Tandem's nonstop system. *Proceedings of the ACME'81 Conference*, page 245, 1981.

[41] K. C. Yeager. The MIPS R10000 superscalar processor. *IEEE Micro,* volume 16, issue 2, pages 28-40, April 1996.

[42] V.K. Reddy, S. Parthasarathy and E. Rotenberg. Understanding prediction-based partial redundant threading for low-overhead, high-Coverage fault tolerance. *12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-12)*, pp: 93-94, October 2006.

[43] V.K. Reddy, A. Al-Zawawi and E. Rotenberg. Assertion-based microarchitecture design for improved fault tolerance. *24th International Conference on Computer Design (ICCD-24)*, October 2006.

[44] V.K. Reddy and E. Rotenberg. Inherent Time Redundancy (ITR): Using program repetition for low-overhead fault tolerance. *37th International Conference on Dependable Systems and Networks (DSN-37),* pp: 307-316, June 2007.

[45] S. Borkar. Designing reliable systems from unreliable components: The challenges of transistor variability and degradation. *IEEE Micro,* volume 25, issue 6, pages 10-16, 2005.

[46] P. Hazucha, T. Karnik, J. Maiz, S. Walstra, B. Bloechel, J. Tschanz, G. Dermer, S. Hareland, P. Armstrong, and S. Borkar. Neutron soft error rate measurements in a 90-nm CMOS process and scaling trends in SRAM from 0.25-/spl mu/m to 90-nm generation. *IEEE Electron Devices Meeting,* pages 21.5.1-21.5.4, 2003.

[47] J. F. Ziegler et al. IBM experiments in soft fails in computer. *IBM Journal of R&D,* volume 40, issue 1, 1998.

[48] E. Normand. Single event upset at ground level. *IEEE Transactions on Nuclear Science,* Volume 43, issue 6, 1996.

[49] S. Michalak, K. W. Harris, N. W. Hengartner, B. E. Takala, and S. A. Wender. Predicting the number of fatal soft errors in Los Alamos National Laboratory's ASC Q supercomputer. *IEEE Transactions on Device and Materials Reliability*, volume 5, issue 3, 2005.

[50] R. C. Baumann. Soft errors in commercial semiconductor technology: Overview and scaling trends. *IEEE 2002 Reliability Physics Tutorial Notes, Reliability Fundamentals,* pages: 121_01.1-121_01.14, April 2002.

[51] T. J. Slegel et al. IBM's S/390 G5 microprocessor design. *IEEE Micro,* volume 19, issue 2 , pages 12-23, 1999.

[52] M. A. Check and T. J. Slegel. Custom S/390 G5 and G6 microprocessors. *IBM Journal of R&D,* volume 43, issues 5/6. 1999.

[53] J. M. Tendler, J. S. Dodson, J. S. Fields, H. Le, and B. Sinharoy. Power4 system microarchitecture. *IBM Journal of R&D,* volume 46, issue 1, 2002.

[54] R. Teodorescu, J. Nakano and J. Torrellas. SWICH: A prototype for efficient cache-level checkpoint and rollback. *IEEE Micro*, volume 26, issue 5, pages 28-40, October 2006.

[55] D. Sorin, M. M. K. Martin and M. D. Hill. Fast checkpoint/recovery to support kilo-instruction speculation and hardware fault tolerance. Tech. Report: CS-TR-2000-1420, Univ. of Wisconsin, Madison. October 2000.

[56] P. Shivakumar and N. P. Jouppi. Cacti 3.0: An Integrated Cache Timing, Power and Area Model. Western Research Lab (WRL) Research Report. 2002.

[57] A. Sodani and G. S. Sohi. Dynamic instruction reuse. *24th International Symposium on Computer Architecture (ISCA-24),* pp: 194:205, 1997.

[58] A. Sodani and G. S. Sohi. An empirical analysis of instruction repetition. *8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-8)*, pp: 35-45, 1998.

[59] A. Mendelson and N. Suri. Designing high-performance and reliable superscalar architectures – The out of order reliable superscalar (O3RS) approach. *30th International Conference on Dependable Systems and Networks (DSN-30),* pp: 478-481, 2000.

[60] M. Franklin, G. S. Sohi and K. K. Saluja. A study of time-redundant techniques for high-performance pipelined computers. *19th International Symposium on Fault-Tolerant Computing (FTCS-19)*, pp. 436:443, 1989.

[61] Illinois Advanced Computing Systems Group. Illinois Verilog Model (IVM). http://www.crhc.uiuc.edu/ACS/tools/ivm/about.html.

[62] D.P. Siewiorek and R. S. Swarz. Reliable Computer Systems: Design and Evaluation. *Digital Press*, 1992.

[63] R. E. Lyons and W. Vanderkulk. The use of triple-modular redundancy to improve computer reliability. *IBM Journal of R&D,* volume 6, issue 2, Page 200 (1962).

[64] Y. Yeh. Triple-triple redundant 777 primary flight computer. *IEEE Aerospace Applications Conference*, volume 1, pages 293–307, February 1996.

[65] J. C. Smolens, B. T. Gold, B. Falsafi, and J. C. Hoe. Reunion: Complexity-effective multicore redundancy. *39th International Symposium on Microarchitecture (MICRO-2006)*, pp 223-234, 2006.

[66] A. Parashar, S. Gurumurthy, and A. Sivasubramaniam. SlicK: Slice-based locality exploitation for efficient redundant multithreading. 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-12), pp: 95:105, 2006.

[67] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, D. I. August, and S. Mukherjee. Software-Controlled Fault Tolerance. *ACM Transactions on Architecture and Code Optimization (TACO)*, December 2005.

[68] J. Chang, G. A. Reis, and D. I. August. Automatic instruction-level software-only recovery methods. *36th International Conference on Dependable Systems and Networks (DSN-36),* pp: 83-92, 2006.

[69] G. A. Reis, J. Chang, and D. I. August. Automatic Instruction-Level Software-Only Recovery Methods. *IEEE Micro,* volume 27, issue 1, January 2007.

[70] J. Chang, G. A. Reis, and D. I. August. Non-uniform fault tolerance. *2nd Workshop on Architectural Reliability (WAR-2)*, December 2006.

[71] C. Wang, H. Kim, Y. Wu, and V. Ying. Compiler managed software-based redundant multi-threading for transient fault detection. *International Symposium on Code Generation and Optimization (CGO-2007),* pp: 244-258, 2007.

[72] S. Palacharla, N. P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. *24th International Symposium on Computer Architecture (ISCA-24)*, pp: 206-218, 1997.

[73] M. Qureshi, O. Mutlu, and Y. Patt. Microarchitecture-based introspection: A technique for transient-fault tolerance in microprocessors. *35th International Conference on Dependable Systems and Networks (DSN-35*), pp: 434-443, 2005.

[74] N. Madan and R. Balasubramonian. Power-efficient Approaches to Redundant Multithreading. *IEEE Transactions on Parallel and Distributed Systems (TPDS), Special Issue on CMPs,* volume 18, issue 8, August 2007.

[75] M. W. Rashid, E. J. Tan, M. C. Huang, and D. Albonesi. Power-efficient error tolerance in chip multiprocessors. *IEEE Micro,* pages 60-70, volume 25, issue 6, 2005.

[76] S. Narayanasamy, A. Coskun, and B. Calder. Predicting faults based on anomalies in speculative execution. *Design, Automation, and Test in Europe Conference & Exhibition,* pp: 1-6, April 2007.

[77] E. Jacobsen, E. Rotenberg, and J. E. Smith. Assigning Confidence to Conditional Branch Predictions. *29th International Symposium on Microarchitecture (MICRO-29)*, pp. 142-152, December 1996.

[78] J. C. Smolens, B. T. Gold, J. Kim, B. Falsafi, J. C. Hoe, and A. G. Nowatzyk. Fingerprinting: Bounding the Soft-Error Detection Latency and Bandwidth. *11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XI),* pp: 224-234, October 2004.

[79] C. L. Chen and M. Y. Hsiao. Error-correcting codes for semiconductor memory applications: A state-of-the-art review. *IBM Journal of R&D,* volume 28, issue 2, page 124, 1984.

[80] N. Quach. High availability and reliability in the itanium processor. *IEEE Micro,* volume 20, issue 5, pages: 61-69, September/October 2000.

[81] E. Fetzer, D. Dahle, C. Little, and K. Safford. The parity protected, multithreaded register files on the 90-nm itanium microprocessor. *IEEE Journal of Solid-State Circuits*, volume 41, issue 1, pages: 246-255, January 2006.

[82] C. McNairy and R. Bhatia. Montecito: A dual-core, dual-thread itanium processor. *IEEE Micro,* volume 25, issue 2, pages: 10-20, March/April 2005.

[83] M. Nicolaidis. Design for soft error mitigation. *IEEE Transactions on Device and Materials Reliability,* volume 5, issue 3, pages 405-418, September 2005.

[84] K. Wilken and J. P. Shen. Continuous signature monitoring: Low-cost concurrent detection of processor control errors. *IEEE Transactions on Computer-Aided Design*, volume 9, issue 6, June 1990.

[85] N. Saxena and E. J. McCluskey. control-flow checking using watchdog assists and extended-precision checksums. *IEEE Transactions on Computers,* volume 39, issue 4, April 1990.

[86] T. Bressoud and F. Schneider. Hypervisor-based fault tolerance. *ACM Transactions on Computer Systems*, pages: 80-107, 1996.

[87] C. Constantinescu. Trends and challenges in VLSI circuit reliability. *IEEE Micro,* volume 23, issue 4, pages: 14-19, July-August 2003.

[88] N. Seifert and N. Tam. Timing vulnerability factors of sequentials. *IEEE Transactions on Device and Materials Reliability,* volume 4, issue 3, September 2004.

[89] Y. Tosaka et al. Comprehensive Study of Soft Errors in Advanced CMOS Circuits with 90/130 nm Technology. *IEEE International Electronic Devices Meeting*, pages: 941-944, December 2004.

[90] E. Dupont, M. Nicolaidis, and P. Rohr. Embedded robustness IPs for Transient-Error-Free ICs. *IEEE Design and Test of Computers*, pages: 56-70, 2002.

[91] A. Shye, T. Moseley, V. Janapa Reddi, J. Blomstedt, and D. A. Connors. Using Process-Level Redundancy to Exploit Multiple Cores for Transient Fault Tolerance. 37th *International Conference on Dependable Systems and Networks (DSN-37),* pp: 297:306, June 2007.

[92] T. Luck. Machine check recovery for linux on itanium processors. *Proceedings of the Linux Symposium,* pages: 313-320, July 2003.

[93] D. A. Patterson, G. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (RAID). *Proceedings of 1988 ACM SIGMOD Conference*, pp: 109–116, June 1988.

[94] T. May and M. Woods. Alpha-particle-induced soft errors in dynamic memories. *IEEE Transactions on Electron Devices*, volume 26, issue 2, pages: 2-9, 1979.

[95] T. J. Dell. A white paper on the benefits of chipkill-correct ECC for PC server main memory. *IBM Microelectronics Division Whitepaper,* November 1997.

[96] IBM. Enhancing IBM netfinity server reliability: IBM chipkill memory. *IBM Whitepaper,* February 1999.

[97] C. Keltcher et al. The AMD hammer processor core. *14th Hot Chips Syposium (Hot Chips-14),* 2002.

[98] C. Keltcher, K. McCgrath, A. Ahmed, and P. Conway. The AMD opteron processor for multiprocessor servers. *IEEE Micro,* volume 23, issue 2, pages: 66-76, March-April 2003.

[99] S. S. Mukherjee, J. Emer, T. Fossum, and S. K. Reinhardt. Cache scrubbing in microprocessors: myth or necessity? *10th Pacific Rim International Symposium on Dependable Computing,* pp: 37-42, March 2004.

[100] D. Ernst, N. S. Kim, S. Das, Sanjay Pant, T. Pham, R. Rao, C. Ziesler, D. Blaauw, T. Austin, T. Mudge, and K. Flautner. Razor: A Low-Power Pipeline Based on Circuit-Level Timing Speculation. *36th International Symposium on Microarchitecture (MICRO-36),* pp: 7-18, December 2003.

[101] D. Ernst, S. Das, S. Lee, D. Blaauw, T. Austin, T, Mudge, N. S. Kim, and K. Flautner. Razor: circuit-level correction of timing errors for low-power operation. *IEEE Micro*, volume 24, issue 6, November 2004.

[102] T. Austin, D. Blaauw, T. Mudge, and K. Flautner. Making typical silicon matter with Razor. *IEEE Computer*, volume 37, issue 3, pages: 57-65, March 2004.

[103] X. Vera, O. Unsal, and A. Gonzalez. X-Pipe: An adaptive resilient microarchitecture for parameter variations. *Workshop on Architectural Support for Gigascale Integration,* June 2006.

[104] S. Mitra, N. Seifert, M. Zhang, Q. Shi, and K. S. Kim. Robust system design with built-in soft-error resilience. *IEEE Micro,* volume 38, issue 2, pages 43-52, February 2005.

[105] P. K. Lala. Self-checking and fault-tolerant design. Morgan Kaufmann Publishers Inc, ISBN:0-12-434370-8, 2000.

[106] J. E. Smith and P. Lam. A theory of totally self-checking system design. IEEE Transactions on Computers, volume C-32, issue 9, pages: 831-844, September 1983.

[107] C. Bolchini and D. Sciuto. A state encoding for self-checking finite state machines. Design Automation Conference (DAC 1995), pp: 711-716, 1995.