ABSTRACT

VIJAYA KUMAR, SOUNDER RAJAN. RTL Design and Analysis of a Fault Check Regimen for Superscalar Processors. (Under the direction of Dr. Eric Rotenberg.)

Although technology scaling over the last few decades has yielded a tremendous boost to the performance of microprocessors, the high logic density on these billion-transistor chips has reduced their reliability. Owing to the increased probability of being affected by faults in the field, microprocessors need built-in fault detection and recovery mechanisms. Conventional fault detection mechanisms based on space, time or information redundancy are too expensive to be incorporated into commodity microprocessors. Novel approaches to protect microprocessors are necessary, that offer broad fault coverage with low implementation overhead.

This thesis extends recent work, which proposed to detect faults in superscalar processors using a regimen of low cost microarchitecture-level checks. This approach comprises of a suite of checks that make use of microarchitectural insights to detect arbitrary faults in large units. A high-level analysis of this approach has shown that a small number of such checks are capable of providing broad fault coverage in superscalar processors.

This thesis, firstly, takes into account the cons of implementing the already proposed regimen and develops a new regimen of microarchitecture-level fault checks. The new regimen is aimed at detecting faults in physical register tags throughout the entire pipeline, thereby covering critical units, including the register rename logic, the issue logic and the retirement
logic. The new regimen contains fault checks that are more effective due to their end-to-end fault detection capability, beginning at the point of generation of physical register tags all the way until their consumption by the instruction. The new fault checks are also, comparatively, more efficient from an implementation standpoint and have a smaller overhead on the baseline core. The second contribution of this thesis is the RTL implementation of the new fault check regimen and its incorporation into a fully synthesizable RTL model of a superscalar processor. In addition, a fault injection infrastructure is developed and integrated into the RTL model of the processor to carry out fault injection experiments and to analyze the effect of faults on the targeted structures. Extensive fault injection experiments carried out on targeted structures show that the new regimen covers more than 99% of non-masked faults. The RTL model was also used to determine the implementation overhead of the fault check regimen on the baseline core, which was found to be less than 6%.
RTL Design and Analysis of a Fault Check Regimen for Superscalar Processors

by

Sounder Rajan Vijaya Kumar

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the degree of
Master of Science

Computer Engineering

Raleigh, North Carolina
2010

APPROVED BY:

_______________________________
Dr. Eric Rotenberg
Committee Chair

_______________________________
Dr. Gregory Byrd

_______________________________
Dr. Yan Solihin
DEDICATION

To My Parents…
BIOGRAPHY

Sounder Rajan V was born in 1987 in Bangalore, India. He underwent his schooling at S. Cadambi Vidya Kendra, Bangalore and at Seshadripuram Composite Pre-University College, Bangalore. He graduated with a Bachelor of Technology in Electrical and Electronics Engineering from National Institute of Technology Karnataka, Surathkal, India, in 2008. He started pursuing Master of Science in Computer Engineering at North Carolina State University in Fall 2008. After obtaining his M.S., he will be joining Freescale Semiconductor as a Power Architecture SoC Verification Engineer.
ACKNOWLEDGEMENTS

I would like to thank Prof. Eric Rotenberg for advising me and guiding me with patience and interest, throughout the course of my M.S. study at NC State. His extraordinary teaching abilities and intricate attention to detail have constantly guided me to be on the right path, and will continue to be my motivation to excel. It surely was an honor working with him. I would like to thank Dr. Gregory Byrd and Dr. Yan Solihin for serving as members of my committee and for their positive inputs during and after my defense. I would also like to thank my colleagues in CESR - Muawya Al-Otoom, Niket Choudhary, Salil Wadhavkar, Hiran Mayukh, Jayneel Gandhi and Tanmay Shah for all their valuable and timely help. I would like to convey special thanks to Hiran, as my roommate, for bugging me every so often regarding the progress of my thesis work.

I would like to express my deepest gratitude to my parents for their constant motivation and unwavering support throughout my education. I would also like to thank all my relatives and friends, in India and in the US, for all their cooperation and support.
# TABLE OF CONTENTS

LIST OF TABLES .................................................................................................................. viii
LIST OF FIGURES ................................................................................................................ ix

Chapter 1 : Introduction ........................................................................................................ 1
  1.1 Thesis Overview ........................................................................................................ 4
  1.2 Organization ............................................................................................................. 8

Chapter 2 : Background and Related Work ........................................................................ 9
  2.1 Effect of a Bit Flip On Program Outcome ................................................................. 10
  2.2 Fault Modeling Strategies ....................................................................................... 13
  2.3 Microarchitectural Fault Detection Techniques ...................................................... 15
  2.4 Related Work ........................................................................................................... 16

Chapter 3 : Baseline Superscalar Processor ..................................................................... 19
  3.1 Flow of Register Tags in a Superscalar Processor ...................................................... 20
    3.1.1 Generation of physical source and destination register tags ......................... 20
    3.1.2 Propagation of source and destination register tags ................................... 22
    3.1.3 Consumption of source register tags ............................................................ 24
    3.1.4 Consumption of destination register tags ...................................................... 24
  3.2 Break-up of Pipeline Register Contents .................................................................... 26
  3.3 Recovery Mechanism in the Baseline Superscalar Processor .................................... 29
    3.3.1 Branch misprediction recovery .................................................................... 29
    3.3.2 Load violation recovery .............................................................................. 30

Chapter 4 : Fault Vulnerability of Register Tags in a Superscalar Processor ................. 32
  4.1 Vulnerabilities in the Register Rename Stage ......................................................... 32
  4.2 Vulnerabilities in the Dispatch Stage ....................................................................... 34
  4.3 Vulnerabilities in the Issue Stage ............................................................................ 35
  4.4 Vulnerabilities in the Register-Read Stage .............................................................. 36
  4.5 Vulnerabilities in the Execute Stage ....................................................................... 37
4.6 Vulnerabilities in the Writeback Stage ........................................................................... 37
4.7 Vulnerabilities in the Retire Stage ................................................................................ 37

Chapter 5 : Proposed Microarchitecture-level Fault Check Regimen for a Superscalar Processor ................................................................. 39

5.1 Register Re-renaming Techniques ................................................................................ 39
  5.1.1 Source register re-renaming ..................................................................................... 40
  5.1.2 Destination register re-renaming ............................................................................. 46
5.2 Previous Mapping Check ............................................................................................. 50
5.3 Register Readiness Checks ........................................................................................ 54
  5.3.1 Ready bit status check I .......................................................................................... 55
  5.3.2 Ready bit status check II ........................................................................................ 56
5.4 Watchdog Timer ........................................................................................................... 63
5.5 A Unified Fault Check Regimen ................................................................................... 64

Chapter 6 : Fault Injection Methodology ........................................................................... 67

6.1 Fault Injection Environment ........................................................................................ 68
6.2 Analysis of Possible Outcomes .................................................................................... 71

Chapter 7 : Fault Injection Results ..................................................................................... 75

7.1 Rename-Dispatch Pipeline Register ............................................................................ 76
7.2 Issue-Register Read Pipeline Register ........................................................................ 78
7.3 Register Read-Execute Pipeline Register ..................................................................... 79
7.4 Execute-Writeback Pipeline Register ......................................................................... 80
7.5 Architectural Map Table ............................................................................................... 81
7.6 Rename Map Table ...................................................................................................... 86
7.7 Speculative Free List .................................................................................................... 88
7.8 Issue Queue Payload ................................................................................................... 89
7.9 Issue Queue Source Ready Bits .................................................................................. 90
7.10 Issue Queue CAM ....................................................................................................... 91
7.11 Active List ................................................................................................................... 92
LIST OF TABLES

Table 3-1 Microarchitectural configuration of RTL model ............................................................... 20
Table 3-2 Breakup of various fields in instruction packets stored in pipeline latches and other structures ............................................................................................................................... 27
Table 4-1 Effect of fault manifestations in Register Rename stage ................................................. 34
Table 4-2 Effect of fault manifestations in the Issue stage ................................................................. 35
Table 5-1 Fault coverage of pipeline flip-flops by the fault check regimen .................................... 65
Table 5-2 Fault coverage of RAMs and CAMs by the fault check regimen .................................. 65
Table 6-1 Fault number assignment to various structures ............................................................... 68
Table 7-1 Relevant benchmark measurements at the end of 100,000 clock cycles ...................... 76
LIST OF FIGURES

Figure 2-1 Possible outcomes of a bit flip from a microarchitectural standpoint. Boxes with bold outlines represent the outcomes ................................................................. 12
Figure 3-1 Pipeline stages of a canonical superscalar processor .................................................. 19
Figure 3-2 Flow of register tags in a superscalar processor .......................................................... 25
Figure 3-3 Break-up of pipeline register contents ........................................................................... 26
Figure 4-1 Rename stage with vulnerable combinational logic blocks ........................................... 33
Figure 4-2 Issue stage with vulnerable combinational logic blocks ............................................ 36
Figure 5-1 (a) Implementation of source re-renaming technique, and (b) Fault coverage of source re-renaming technique ...................................................................................... 43
Figure 5-2 (a) Implementation of destination re-renaming, and (b) Fault coverage of destination re-renaming .............................................................................................................. 48
Figure 5-3 Corner cases that evade destination re-renaming but detected by RBS II .......... 60
Figure 5-4 Issue and Register Read pipeline stages with RBS I and RBS II fault checks .... 62
Figure 5-5 Register tag flow after structural additions (in bold outline) as part of incorporating the proposed fault check regimen ................................................................. 66
Figure 6-1 Fault injection environment .......................................................................................... 70
Figure 7-1 Fault injection results for Rename-Dispatch pipeline register (starred fields ‘*’ were added as part of the unified fault check regimen) .................................................. 78
Figure 7-2 Fault injection results for Issue-Register Read pipeline register ................................... 79
Figure 7-3 Fault injection results for Register Read-Execute pipeline register .............................. 80
Figure 7-4 Fault injection results for Execute-Writeback pipeline register (starred fields ‘*’ were added as part of the unified fault check regimen) .................................................. 81
Figure 7-5 Fault injection results for Architectural Map Table ..................................................... 82
Figure 7-6 Register usage and Fault distribution for the bzip benchmark ................................. 83
Figure 7-7 Fault injection results for Rename Map Table ............................................................ 87
Figure 7-8 Fault injection results for Speculative Free List ......................................................... 88
Figure 7-9 Fault injection results for Issue Queue payload ......................................................... 90
Figure 7-10 Fault injection results for the source ready bits in Issue Queue ............................... 91
Figure 7-11 Fault injection results for the physical source register CAM in Issue stage ...... 91
Figure 7-12 Fault injection results for Active List ........................................................................ 93
Figure 8-1 Area breakup of pipeline stages in a superscalar processor ........................................ 97
Chapter 1: Introduction

A negative side effect of deep technology scaling, which has recently started to dominate the design decisions of microarchitects and circuit designers, is the degraded reliability of microprocessors. The high logic density in today’s billion-transistor chips not only requires high verification and validation effort but also requires effective mechanisms to detect and recover from faults in the field. Faults occurring in the field could be either due to permanent device failures or due to single event upsets (aka soft errors) caused by energetic particles in the atmosphere.

Traditional fault protection techniques are mostly based on redundancy in space, time or information. Space redundancy based techniques replicate hardware to detect faulty behavior, and could involve replicating specific components such as functional units, or replicating the processor as a whole. An example of space redundancy would be the IBM S/390 G5 microprocessor [1] [2], which duplicates the fetch, decode and execution units to detect faults. An extreme case of space redundancy is Triple Modular Redundancy (TMR), which involves executing the same program on three independent processors and comparing their outputs to detect and mask faults. Fault detection mechanisms based on time redundancy exploit the existing hardware to run multiple instances of the same program and compare their outcomes. Redundant Multithreading (RMT) is a technique based on time redundancy that has been proposed in various forms [3] [4] [5] [6]. Information redundancy techniques,
such as parity and error correction codes (ECC), typically add redundant bits to encode data in various memory structures and datapaths [7].

Clearly, these techniques are very expensive to be incorporated into commodity microprocessors due to their high overhead in terms of area, power and performance. Space redundancy based techniques have high area and power overhead, while time redundancy based methods have performance and power overhead. Information redundancy techniques mostly incur area and performance overhead. Hence, contemporary research work is focused on developing innovative cost-effective techniques for commodity microprocessors that have high fault coverage and low overhead to the baseline microprocessor.

One example of an efficient fault detection methodology for commodity processors was recently proposed by Reddy et al. [8]. They proposed a regimen of microarchitecture-level fault checks capable of detecting faults in superscalar processors. They observed that knowledge of the processor microarchitecture could be effectively exploited to develop a suite of checks added at key locations within the microprocessor. These checks are capable of detecting faults in large units, thereby providing broad fault coverage with low overhead. For example, the previous mapping check that was proposed in [REF], exploits the microarchitectural insight that an instruction perceives the exact same register mappings in the Rename Map Table (RMT) and in the Architectural Map Table (AMT) during its Register renaming stage and Retirement stage respectively. Thus, asserting that the RMT mapping of the destination register recorded in the Renaming stage matches its AMT.
mapping during instruction retirement, can detect faults in both the RMT as well as the AMT. The authors claimed that, on average, 83\% of non-masked\(^1\) faults affecting the processor were detected by the proposed fault check regimen.

Although the regimen proposed by Reddy et al. \cite{Reddy2012} provide a good coverage of faults, their experiments were performed on a C++ cycle-accurate simulator. Results obtained by experiments on a high-level simulator are hard to map to the actual hardware due to the lack of direct correspondence between faults injected in the high-level simulator and those affecting logic gates. Moreover, the proposed techniques have not been comprehensively analyzed from a hardware perspective, in terms of their overhead to the underlying processor.

This thesis builds upon the idea of the proposed regimen and makes two key contributions. Firstly, it develops a new fault check regimen capable of providing end-to-end protection to physical register tags, thereby covering all pipeline stages involved in their generation, propagation and consumption, until instruction retirement. The new regimen offers broader fault coverage than the one proposed earlier and is efficient from an implementation perspective. The second contribution is the RTL design and analysis of the new fault check regimen, and its incorporation to the RTL model of a superscalar processor. A fault injection infrastructure has been developed to inject faults into the RTL model of the processor, thereby enabling RTL fault analysis, which provides a better estimate of the fault coverage of

\(^1\) A fault is said to be masked when it is not propagated to a higher abstraction level. A discussion on fault masking can be found in Chapter 2.
the regimen, compared to a high-level fault analysis. Also, synthesis of the modified RTL model has been carried out to obtain a good estimate of the implementation overhead of the regimen to the baseline core.

The current work concentrates only on fault detection. Recovery from faults has been left for future work.

1.1 Thesis Overview

Since this thesis directly builds upon prior research work conducted by Reddy et al. [8], it is beneficial to briefly look at their work. Their fault check regimen can be broadly classified into the following categories based on their coverage:

1) Checks that perform register name authentication: The previous mapping check (RNA1), the writeback state check (RNA2) and the register consumer counter (CC) check are involved with detecting faults in the entire register renaming logic and those affecting the communication of correct register information between various pipeline stages.

2) Checks that protect the issue logic: The Timestamp based Assertion Check (TAC) asserts the time-orderliness within a data dependence chain, thereby detecting premature issue of instructions.

3) Checks that protect the static information in an instruction packet: Inherent Time Redundancy (ITR) exploits the repetitive nature of instructions within a program to
detect faults in information that remains constant over multiple instances of an instruction.

4) Miscellaneous checks such as the watchdog timer, sequential PC check and the BTB Verify check which aim at covering some of the faults not covered by the other specialized checks.

This thesis mainly focuses on categories 1 and 2, and aims at developing a new regimen that has a comprehensive end-to-end coverage of faults affecting:

a) the correctness of physical register tags used for instruction execution, and
b) the timing associated with the reading and writing of physical registers during instruction execution

The new regimen takes into account the drawbacks of some of the checks already proposed, and introduces new microarchitecture-level checks that have broader coverage and smaller implementation complexity. The new regimen proposed in this thesis comprises of the following microarchitecture-level checks:

1) Source re-renaming: The earlier proposed Register Consumer Counter (CC) check for detecting faults in physical source register tags proved to be expensive to implement in hardware, since it requires maintaining counters for each of the 128 physical registers, along with the peripheral logic for incrementing and decrementing certain counters every cycle. Moreover, counters corresponding to instructions in
mispredicted paths had to be reset, which resulted in an expensive recovery from branch mispredictions, eventually degrading the IPC. From a coverage standpoint, the CC check detected faults in physical source register tags only after register renaming. In the new regimen, the CC check has been replaced by the source re-renaming technique, whereby the logical source registers of instructions are redundantly renamed for a second time during instruction retirement. This technique, thus, provides a comprehensive end-to-end coverage of faults in physical source register tags occurring during and after register renaming.

2) Destination re-renaming: Similar to source re-renaming, this technique involves redundantly renaming the logical destination registers of instructions during retirement. This technique effectively detects faults affecting physical destination register tags throughout the processor pipeline.

3) Ready bit status checks: The Timestamp based Assertion check (TAC) has been replaced by two ready bit status (RBS) checks that serve the same purpose as TAC with a much smaller overhead. The RBS checks not only detect premature issue of instructions, but also prove effective in detecting faults in various other structures, such as the Speculative Free List.

4) Previous mapping check: This check, part of the proposed regimen in [9], involves comparing the previous mapping in the Rename Map Table (RMT) recorded at register rename stage with the corresponding mapping in the Architectural Map Table
(AMT) during retirement. This check is effective in detecting some faults in the logical destination registers, the RMT and the AMT.

5) Watchdog timer: A watchdog timer forms an effective low cost technique of detecting deadlocks within the microprocessor.

For fault injection, the current work assumes a single fault model represented by a bit-flip. A hierarchical fault injection infrastructure has been developed that has a perl script at the top level, the Verilog simulation testbench at the intermediate level, and the actual Verilog RTL model at the lowest level. An experiment consists of a set of trials, each trial characterized by a fault location and a clock cycle for fault injection, both of which are generated by the top level script and propagated to the RTL model for fault injection. Locations for fault injection include all pipeline flip-flops holding physical and logical register tags, in addition to RAM and CAM structures involved in register renaming, out-of-order issue and instruction retirement, including Rename Map Table, Architectural Map Table, Speculative Free List, Issue Queue payload RAM, the Issue Queue CAM and the Active List. The cycle number for fault injection lies in a predetermined range of clock cycle count from the start of simulation.

Extensive fault injection experiments on the various targeted structures revealed an overall coverage of the new fault check regimen to be greater that 99% of non-masked faults. Fault injection experiments performed on most of the critical structures such as the Rename Map Table and the Issue Queue CAM yielded 100% fault coverage. However, a small percentage (< 1%) of faults in the Speculative Free List and the Architectural Map Table remained
undetected. The regimen also provided more than 99% coverage of faults affecting relevant fields in pipeline registers. Specific scenarios that result in undetected faults are discussed in Chapter 7. Synthesis of the RTL model incorporating the fault check regimen yielded an area overhead of less 6% on the baseline core.

1.2 Organization

This document is organized as follows. Chapter 2 contains a brief background on soft errors, their effect on program outcome and various fault modeling strategies. It also discusses common microarchitectural techniques to detect faults in superscalar processors and related work in more detail. Chapter 3 introduces the baseline superscalar core and explains register tag management and recovery mechanisms in the baseline superscalar RTL model. Chapter 4 dissects each pipeline stage for vulnerabilities in the combinational logic blocks dealing with register tags and traces faults affecting them to the nearest pipeline register. Chapter 5 describes the proposed microarchitecture-level fault detection techniques, culminating with a unified fault check regimen. Chapter 6 explains the fault injection infrastructure and methodology followed in the current work. Chapter 7 analyzes the coverage results of the implemented fault check regimen, obtained by fault injection on the targeted structures. Chapter 8 presents RTL synthesis results of the fault check regimen, its overhead on the baseline core and its silicon area coverage of the baseline core. Chapter 9 concludes the thesis and discusses future work.
Chapter 2 : Background and Related Work

Today’s high-density microprocessors can be affected by a number of faults, ranging from bugs in the design stage all the way to faults in the field due to latent defects and soft errors. While latent defects are easier to detect due to their permanent nature, soft errors pose a greater challenge due to their transient behavior. Soft errors are generally radiation-induced and arise due to the interaction of silicon with energetic particles, such as alpha particles from packaging material and neutrons from the atmosphere. These energetic particles cause charge to be accumulated in transistors. When the accumulated charge crosses a threshold, called Qcrit for “critical charge” [10], it may invert the state of a logic device, resulting in faulty circuit operation.

Soft errors and their impact on memory and logic circuits have been extensively studied since their discovery in 1979 [10] [11]. Though initial studies were mainly conducted on memory cells, the development of effective memory protection strategies led to the shift of focus to protecting dynamic and static logic. Tools and methodologies have been developed to model the effect of soft errors on circuit operations [12] [13]. Baumann [14] [15] quantified the effects of technology scaling on soft error rate (SER) and concluded that failure in time per bit saturates for memory devices but the system SER continues to increase as the number of susceptible structures in a design grows. He predicted that sequential logic SER would become the reliability limiter once SRAMs were protected. The factors that need to be considered to completely model the propagation of a soft error event to a sequential structure
holding the final logical state of a device are being researched upon. Nguyen et al. [16] describe a systematic approach to estimate the SER of a typical high-performance processor based on the concepts of nominal failure in time (FIT), time derating and logic derating.

### 2.1 Effect of a Bit Flip On Program Outcome

Microprocessors have different levels of abstraction - transistors that form circuits, circuits that form logic gates and storage devices, and finally the gates and storage devices that comprise the processor itself. Fault masking can occur at any of these abstraction levels. A particle strike that causes enough charge accumulation on a transistor is logically seen at the gate level as a bit flip. Masking occurs at the lowest abstraction level when such a bit flip does not propagate forward to a flip-flop. Even if a faulty bit is latched, whether it affects the outcome of the program depends on various factors such as, the location of bit-flip and the dynamic state of the processor when the fault occurs. Masking may occur

Figure 2-1 shows the various possible outcomes due to a bit-flip, from a microarchitectural perspective. As indicated by the figure, clearly a bit flip that occurs in a transistor does not necessarily result in an erroneous program outcome. If the faulty bit is not latched due to electrical or logical masking, then the bit flip is benign (outcome 1). If the faulty bit is latched but is not capable of affecting the program outcome, then this too is a benign/masked case of a bit flip (outcome 2). This happens if, for example, the fault occurs in the branch predictor logic, and hence does not affect program outcome but can affect performance (IPC) of the processor. If the faulty latch is capable of erroneous program outcome, then whether
the fault is harmful or not depends on the timing and the dynamic state of the processor (outcome 3). If, for example, the fault occurs during the execution of instructions in a mispredicted path, the fault may be masked eventually. On the other hand, if the fault manifests in a suitable scenario as to cause an erroneous program outcome, then the fault detection and recovery logics come in to play. If the processor does not have logic to handle faults, then the fault will result in silent data corruption (SDC) (outcome 4). If the processor has only fault detection logic but no fault recovery logic, then it results in a detected unrecoverable error (DUE), which normally results in a restart of program execution (outcome 5). However, the presence of recovery logic will successfully resolve the fault and prevent the occurrence of an erroneous program outcome without restarting the program (outcome 6). In this thesis, faults that result in outcomes 1 to 3, where they never reach the architectural state are considered to be ‘masked’ faults, whereas those that reach the architectural state directly or indirectly (e.g., via a committed instruction that uses a faulty register tag) but do not cause corruption are termed ‘benign’.
Figure 2-1 Possible outcomes of a bit flip from a microarchitectural standpoint. Boxes with bold outlines represent the outcomes.
2.2 Fault Modeling Strategies

Fault analysis can be performed at various abstraction levels by taking into account the various factors that come into play at each level. Protection techniques exist at each abstraction level to mitigate the effects of faults and reduce the probability of fault propagation to higher abstraction levels. While device and gate level protection techniques have low area overhead, their coverage is low when compared to the extreme case of triple modular redundancy (TMR) supported in mainframe systems, which involves running copies of the same program on three different microprocessors to detect and mask any errors. Consequently, TMR offers much higher coverage but at the cost of area and power. A cost-effective protection mechanism finds a balance between the coverage it provides and the overhead it results in.

The current study is concerned with modeling faults at the microarchitectural level, owing to the capability to design effective microarchitecture-level fault detection mechanisms that have good coverage without significantly high overheads. Architectural Vulnerability Factor (AVF) [17] is a commonly used metric by microarchitects to identify candidates at the microarchitecture level that require fault protection. AVF is the probability that a state change in a device leads to an error at the architectural level and hence, AVF of various structures in a microprocessor can vary from 0% (e.g., Branch History Table) to 100% (e.g., Program Counter). Mukherjee et al. [17] discuss three approaches to estimate the AVF of different processor structures – 1) analytical models using Little’s Law [18], 2) ACE analysis
in a performance model [19], and 3) experimental approach by statistical fault injection (SFI).

AVF can be computed experimentally using a simulation model, which could either be a high level abstract representation of the microprocessor, often written in a high level language such as C or C++, or the RTL model, which encompasses the actual architectural definition and timing of the processor. The obvious advantage of using an RTL model is the access it provides to the detailed processor state and various microarchitecture-level structures. Statistical Fault Injection (SFI) methodology is used in conjunction with an RTL model to inject faults into different RTL states, such as pipeline registers, and observe the effect of faults over a period of time. However, a major disadvantage of RTL simulation is the time it takes as compared to that of a high-level performance model. The effect of a fault in certain structures may not show up for millions of cycles. Moreover, the simulations must include a number of benchmarks and must potentially span over millions of transistors on the chip. Nevertheless, experiments have shown that for structures such as pipeline registers, the latency for a fault to manifest is comparatively smaller and is in the order of hundreds of clock cycles, because of the relatively small lifetime of data in such structures. All structures that are targeted for protection in the current work belong to this category and hence are analyzed solely using the RTL model of a microprocessor.
2.3 Microarchitectural Fault Detection Techniques

Error coding techniques such as parity, single error correction (SEC) codes and cyclic redundancy check (CRC) codes, are commonly used today for protecting data in large, dense structures such as RAMs. These coding schemes typically add redundant check bits to a set of data bits so as to be able to detect or even correct errors in the data at a later point in time. Faults in execution units can be detected using arithmetic codes such as residue codes. Residue codes are particularly attractive since they cover a wide variety of arithmetic operations. As an example, IBM’s recent mainframe microprocessor z6 incorporates ECC on second and third level caches, store buffers, parity on all arrays and register files, parity or residue checking on data/address/execution flow, with over 20,000 error checkers in chip [7].

Fault detection by redundant execution has been in use for a long time, especially in systems that demand very high fault tolerance. This technique is used mainly because it provides much higher coverage, typically across an entire processor chip, when compared to error coding techniques, which provide coverage over individual hardware structures inside the chip. Lockstepping and Redundant Multithreading (RMT) are two common schemes that are used for redundant execution. Lockstepping involves maintaining the exact same microarchitectural state across the redundant copies every clock cycle, whereas RMT checks for output correctness across the redundant copies only during the commit of every instruction. Different implementations of each of these schemes can be found in today’s server-grade and mainframe microprocessors. Processors that incorporate Lockstep implementations include HP’s Himalaya [20] and IBM’s G5 [2]. HP’s NonStop® Advanced
Architecture uses a form of RMT [21]. RMT can either be implemented in a single processor core using the underlying architecture for Simultaneous Multithreading (SMT) [3] or in a chip multiprocessor (CMP), where the redundant threads execute on different processor cores on the same chip [22]. DIVA [23] incorporates a specialized in-order checker core that acts as one of the redundant threads.

The implementation overhead of error detection and correction codes as well as redundant execution, whether space-based or time-based, often restricts their usage in commodity microprocessors. Constant research is being carried out to assess the vulnerability of today’s commercial microprocessors and devise novel, cost-effective techniques to protect the critical structures in a contemporary superscalar processor.

2.4 Related Work

Significant amount of work based on Statistical Fault Injection has been carried out to determine the vulnerability of contemporary high performance microprocessors. Wang et al. [24] examined the effects of transient faults on a modern out-of-order microprocessor similar to the Alpha 21264 or AMD Athlon. They conducted SFI on a detailed Verilog model to investigate the vulnerability of the various structures and pipeline latches within the processor core. Their results were categorized based on the region of fault injection (latch+RAM or only latches) and based on the outcome of fault injection (microarchitecture state match, termination, silent data corruption, or gray area). It was found that about 12% of latch+RAM faults and about 9% of latch faults resulted in failures. Moreover, it was also
found that among the RAMs, the architectural map table, the rename map table, and the speculative free list are among the most vulnerable structures. Naturally, among the latches, register information (termed as physical register file pointers in [24]) were found to be vulnerable too, since they are directly associated with the above structures. The same authors also came up with a symptom based fault detection technique called ReStore, where they claim that the occurrence of events such as exceptions and errant control flow serve as symptoms of transient faults, and use architectural checkpoints to restore the system to an earlier point in time [25]. However, the effectiveness of this technique largely depends on the granularity of system checkpoints. Higher the number of checkpoints, smaller is the performance impact but larger is the area overhead. Nguyen et al. [16] performed SFI experiments on an Itanium-class processor to derive the AVF for latches in the processor and found that architectural registers and global control signals such as reset, flush, and stall signals are among the most sensitive to faults. However, since these control signals constitute only a fraction of the total number of bits, they do not impact the overall failure rate significantly.

Meixner et al. [26] propose to build the data flow graph for basic blocks in a program at run time and compare its signature with a statically generated signature by the compiler. An extension of this work [27] uses this technique along with other run-time techniques to detect errors based on invariants in a simple in-order processor. This high-level error detection approach that broadly checks control flow, data flow, computation accuracy and memory interaction, has a high coverage and a modest hardware cost for a simple processor.
However, it requires an additional instruction added to the ISA to pass on signatures to the hardware, and needs the support of the compiler to compute and embed basic block signatures into the program binary.

Carretero et al. [28] developed a signature-based protection mechanism to protect the control logic involved in register data flow. They also integrate the signature-based mechanism with a residue checking scheme to protect the functional units and the physical register file. One of the disadvantages of their approach is that it cannot detect faults if a wrong register is assigned to an instruction, in the first place, since the signature of this wrong register will be consistent throughout. This could happen if, for example, a fault occurs in the address decoding logic of the rename map table. In other words, this approach does not verify the dataflow dependencies among instructions in a program.

Li et al. [29] contest that modeling faults in combinational logic at the latch level is not accurate enough to study the effects of gate-level faults at the system level, especially for permanent and intermittent faults. Since gate-level simulations are very slow to be performed on large systems, they proposed SWAT-Sim, a fault injection infrastructure built on a hierarchical simulation model, where only the faulty component is simulated at a gate-level accuracy but the rest of the simulation takes place at a higher abstraction level. However, since the current work concentrates mainly on pipeline flip-flops and array based structures such as the rename map table, statistical fault injection at the microarchitecture-level is used for modeling transient faults.
Chapter 3 : Baseline Superscalar Processor

The pipeline stages constituting a canonical superscalar processor are shown in Figure 3-1.

![Figure 3-1 Pipeline stages of a canonical superscalar processor](image)

The baseline core that was used for the current thesis has been borrowed from the FabScalar [30] toolset and has the configuration shown in Table 3-1 [31]. For a detailed explanation of the synthesizable RTL model, the interested reader is directed to [31]. Only relevant sections are explained in the current document.

Since the current work concentrates on protecting the flow of register tags within a processor, emphasis is laid on the pipeline stages after the instruction decode stage. A description of pipeline stages and the structures involved in the generation, propagation and consumption of register tags is relevant and useful in understanding their vulnerabilities.
Table 3-1 Microarchitectural configuration of RTL model

<table>
<thead>
<tr>
<th>Stage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch</td>
<td>4-wide, 512 entry BTB, 128 entry bimodal branch predictor, 8-entry RAS, 16-instruction fetch buffer</td>
</tr>
<tr>
<td>Decode</td>
<td>4-wide, SimpleScalar ISA</td>
</tr>
<tr>
<td>Rename</td>
<td>4-wide 32-entry Rename Map Table with 8 read and 4 write ports, 4 shadow map tables (checkpoints)</td>
</tr>
<tr>
<td>Dispatch</td>
<td>4-wide</td>
</tr>
<tr>
<td>Issue</td>
<td>4-wide, 32-entry Issue Queue</td>
</tr>
<tr>
<td>Register-Read</td>
<td>4-wide, 128-entry physical register file with 8 read ports and 4 write ports</td>
</tr>
<tr>
<td>Execute</td>
<td>1 simple ALU, 1 complex ALU, 1 branch ALU, 1 AGEN + 1 port to load-store unit</td>
</tr>
<tr>
<td>Load-Store Unit</td>
<td>16-entry load queue, 16-entry store queue</td>
</tr>
<tr>
<td>Writeback</td>
<td>4-wide</td>
</tr>
<tr>
<td>Retire</td>
<td>4-wide, 128-entry Active List with 4 read and 4 write ports, Architecture Map Table with 4 read and 4 write ports</td>
</tr>
</tbody>
</table>

3.1 Flow of Register Tags in a Superscalar Processor

Figure 3-2 shows the flow of register tags through the pipeline stages post instruction decode, along with the various intermediate structures that hold register tags. The following sections have been divided based on register tag generation, propagation and usage during the lifetime of an instruction i.e., beginning at register rename stage up to the instruction retirement stage.

3.1.1 Generation of physical source and destination register tags

The logical source and destination registers of an instruction are decoded from the instruction packet in the decode stage and passed on to the register rename stage. The renaming logic renames the logical destination of each incoming instruction to a physical register that is popped out by the Speculative Free List (SFL). The SFL is a FIFO that maintains physical
registers that are not currently mapped to any of the logical registers and can be safely used to rename logical destinations of new instructions. This structure is speculative since it does not differentiate between instructions that are in the correct path and those that are in the path that is yet to be resolved. This mapping between the logical destination and the physical destination of an instruction is maintained in a structure called the Rename Map Table (RMT). Thus, every time an instruction is renamed, the entry in the RMT corresponding to the logical destination register is updated with the new physical register mapped to it. In a given cycle, if more than one instruction writes to the same logical destination, only the youngest producer updates the RMT. In the baseline processor implementation, on overwriting the mapping of a register in the RMT, the stale mapping is discarded. However, since the logical destination is required for updating the Architectural Map Table (AMT) during instruction retirement (discussed in Section 3.1.4), it is retained as part of the instruction packet that flows down the pipeline.

The logical source registers of an instruction also undergo renaming in the register rename stage. The rename stage maps each of the logical source registers to a physical register using the Rename Map Table, which contains the latest mapping of each logical register to a physical register, as discussed above. Additionally, since four instructions are renamed every clock cycle, true dependencies among logical source registers and logical destination registers of all prior instructions being renamed in the same cycle, are also resolved.
Thus, the physical register tags corresponding to the source and destination of every instruction are generated in the register rename stage. In the baseline implementation, the logical source registers of instructions are discarded immediately after renaming, and only the physical source registers are passed on to the following pipeline stages. On the other hand, both logical as well as physical destination registers are required until retirement and are part of the instruction packet that flows down the pipeline.

3.1.2 Propagation of source and destination register tags

Dispatch stage follows the Rename stage and forms the boundary between the in-order front-end and the out-of-order backend of the processor. It is involved with dispatching the incoming renamed instructions to the Active List and the Issue Queue (IQ). Active List is a structure that holds incoming instructions in program order until their retirement. The Issue Queue (IQ) holds instructions until they are selected and issued for execution in an out-of-order fashion.

Figure 3-2 shows the information that is dispatched to the AL and the Issue Queue by the dispatch unit. Each of the packets dispatched to the AL contains the logical destination register, the physical destination register and other instruction specific information. This information remains in the AL until the instruction retires. The IQ physically comprises of two independent structures – the IQ payload RAM and the IQ CAM (Content Addressable Memory). Each incoming packet from the dispatch stage is stored in the IQ RAM until the instruction is selected for execution. Additionally, the physical source registers are extracted
and stored in the IQ CAM, which is used for waking up consumer instructions in the Issue Queue.

When an instruction in the IQ is woken up and selected for issue, it proceeds to the Register Read stage for reading out data from the Register File. Also, every instruction’s functional unit type and physical destination register are sent to the RSR module. The RSR is responsible for broadcasting the physical destination of each instruction and waking up its consumer instructions in the IQ. The waking up is carried out two clock cycles before the producer instruction actually completes execution, to ensure back-to-back execution of the producer and consumer instructions. For instructions that take a single clock cycle for execution, their physical destination is broadcast in the same cycle as they enter the register read stage. However, for an instruction that requires more than one clock cycle for execution, the RSR maintains a shift register to store the physical destination of the instruction and broadcasts it only two clock cycles before the completion of execution. Load instructions broadcast their physical destinations only after the data has been fetched from the data cache.

Since the physical destination of every instruction is required until Writeback stage, it is propagated down the pipeline as part of the instruction packet. The physical source registers are required upto the Execute stage in order to extract data off the bypass bus, and hence, are also propagated down the pipeline.
3.1.3 Consumption of source register tags

The physical source register tags are used in the Register-Read pipeline stage following the granting of instruction from the IQ. The source register tags are used to index into the physical register file to read out the corresponding data values that are consumed by the functional units for instruction execution. They are also used in the Execution unit following the Register Read stage to receive data available on the bypass bus. Thus, the Execute stage is the last pipeline stage when the physical source tags are used and hence are discarded without being propagated further to the Writeback stage.

3.1.4 Consumption of destination register tags

In addition to waking up consumers in the IQ, the physical destination register tag of an instruction is used in the Writeback stage for writing into the physical register file and for broadcasting the execution outcome to the consumer instructions.

Additionally, the logical and physical destination register of every instruction are used when the instruction retires or commits. During the commit of an instruction, the Architecture Map Table (AMT) is updated with the latest physical register mapping of the logical destination register. The AMT maintains the committed mappings of logical registers to physical registers. Before replacing the mapping of a logical register in the AMT, its old physical register mapping is read out and released (or freed) by adding it to the Free List.
Figure 3-2 Flow of register tags in a superscalar processor
3.2 Break-up of Pipeline Register Contents

Figure 3-3 shows the break-up of all the pipeline register contents. Register tags form about 12% of the information held in pipeline registers, while static information (e.g., PC, opcode etc.) forms a large percentage (58%).

![Figure 3-3 Break-up of pipeline register contents](image)

Table 3-2 shows the individual fields that constitute each pipeline register following the decode stage. The numbers in parenthesis indicate the size of the field in terms of number of bits.
Table 3-2 Breakup of various fields in instruction packets stored in pipeline latches and other structures

<table>
<thead>
<tr>
<th>Pipeline Register / Structure</th>
<th>Field (Number of bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decode-Rename</td>
<td>Logical Source1 (5),</td>
</tr>
<tr>
<td></td>
<td>Logical Source2 (5),</td>
</tr>
<tr>
<td></td>
<td>Logical Destination (5),</td>
</tr>
<tr>
<td></td>
<td>Immediate data (16),</td>
</tr>
<tr>
<td></td>
<td>FU type (2),</td>
</tr>
<tr>
<td></td>
<td>Opcode (8),</td>
</tr>
<tr>
<td></td>
<td>Program Counter (32),</td>
</tr>
<tr>
<td></td>
<td>Target address (32),</td>
</tr>
<tr>
<td></td>
<td>Other control information (16)</td>
</tr>
<tr>
<td>Rename-Dispatch</td>
<td>Logical Destination (5),</td>
</tr>
<tr>
<td></td>
<td>Physical Destination (7),</td>
</tr>
<tr>
<td></td>
<td>Physical Source 2 (7),</td>
</tr>
<tr>
<td></td>
<td>Physical Source 1 (7),</td>
</tr>
<tr>
<td></td>
<td>Immediate data (16),</td>
</tr>
<tr>
<td></td>
<td>FU type (2),</td>
</tr>
<tr>
<td></td>
<td>Opcode (8),</td>
</tr>
<tr>
<td></td>
<td>Program Counter (32),</td>
</tr>
<tr>
<td></td>
<td>Target address (32),</td>
</tr>
<tr>
<td></td>
<td>Other control information (16)</td>
</tr>
<tr>
<td>Dispatch-Backend</td>
<td>Logical Destination (5),</td>
</tr>
<tr>
<td>Dispatch</td>
<td>Physical Destination (7),</td>
</tr>
<tr>
<td>Backend</td>
<td>Program Counter(32),</td>
</tr>
<tr>
<td>(ActiveList)</td>
<td>Other control information (3)</td>
</tr>
<tr>
<td>(128 entries)</td>
<td></td>
</tr>
<tr>
<td>Dispatch-Issue</td>
<td>Physical Destination (7),</td>
</tr>
<tr>
<td>(Issue Queue Payload RAM)</td>
<td>Immediate data (16),</td>
</tr>
<tr>
<td>(32 entries)</td>
<td>FU type(2),</td>
</tr>
<tr>
<td></td>
<td>Opcode (8),</td>
</tr>
<tr>
<td></td>
<td>Program Counter (32),</td>
</tr>
<tr>
<td></td>
<td>Target address (32),</td>
</tr>
<tr>
<td></td>
<td>Other control information (19)</td>
</tr>
<tr>
<td>Issue Queue CAM</td>
<td>Physical Source 2 (7),</td>
</tr>
<tr>
<td>(32 entries)</td>
<td>Physical Source 2 ready (1),</td>
</tr>
<tr>
<td></td>
<td>Physical Source 1 (7),</td>
</tr>
<tr>
<td></td>
<td>Physical Source 1 ready (1)</td>
</tr>
<tr>
<td>Table 3-2 Continued</td>
<td></td>
</tr>
<tr>
<td>---------------------</td>
<td></td>
</tr>
<tr>
<td><strong>Issue-Register Read</strong></td>
<td>Physical Destination (7), Physical Source 2 (7), Physical Source 1 (7), Immediate data (16), FU type (2), Opcode (8), Program Counter (32), Target address (32), Other control information (29)</td>
</tr>
<tr>
<td><strong>Register Read-Execute</strong></td>
<td>Physical Destination (7), Physical Source 2 (7), Physical Source 1 (7), Immediate data (16), Data (64), FU type (2), Opcode (8), Program Counter (32), Target address (32), Other control information (29)</td>
</tr>
<tr>
<td><strong>Execute-Writeback</strong></td>
<td>Physical Destination (7), Execution Flags (8), Data (32), Program Counter (32), Other control information (22)</td>
</tr>
<tr>
<td><strong>Load Queue</strong> (16 entries)</td>
<td>Physical Source (7), Other control information (12)</td>
</tr>
<tr>
<td><strong>Rename Map Table (RMT)</strong> (32 entries)</td>
<td>Physical Register (7)</td>
</tr>
<tr>
<td><strong>Shadow Map Table (SMT)</strong> (4 SMTs with 32 entries each)</td>
<td>Physical Register (7)</td>
</tr>
<tr>
<td><strong>Speculative Free List</strong> (96 entries)</td>
<td>Physical Register (7)</td>
</tr>
<tr>
<td><strong>Architectural Map Table (AMT)</strong> (32 entries)</td>
<td>Physical Register (7)</td>
</tr>
</tbody>
</table>
Other control information may include any or all of the following fields – LD/ST data size (2), CTI queue tag (4), Branch instruction (1), Store Instruction (1), Load instruction (1), Branch direction (1), Branch mask (4), SMT ID (2), IQ ID (5), AL ID (7), LSQ ID (4).

As can be observed from Table 3-2, most of the bits in pipeline registers are used to store static, instruction-specific information. For example, the rename-dispatch pipeline latch has 63 bits of static information (immediate data, program counter, FU type and opcode). In processors employing good branch predictors, the target address can also be considered static information. Thus, 95 of the total 132 bits (~72%) are dedicated to storing static information. The rest of the bits store dynamic content comprising of register tags and other control information.

3.3 Recovery Mechanism in the Baseline Superscalar Processor

The recovery mechanisms that are implemented in the baseline core play a major role in determining the degree of architectural masking of faults affecting the processor. The following sections describe the handling of branch mispredictions and load violations in the RTL model that is used for current work.

3.3.1 Branch misprediction recovery

The baseline processor incorporates a bimodal branch predictor in the fetch stage. The logic to recover from mispredicted branches mainly consists of four Shadow Map Tables (SMT) that are shown as part of the Rename stage in Figure 3-2. Every time a branch is encountered in the register rename stage, the rename map table is checkpointed in one of the available
SMTs. The head and tail pointers of the Speculative Free List are checkpointed as well, for later recovery if needed. When a branch is resolved in the execute stage, the correctness of the earlier prediction is broadcast to every pipeline stage. If the branch was predicted correctly, then the corresponding SMT is freed and the fetch stage continues fetching instructions in the right path. On the other hand, if a branch was mispredicted, then its corresponding SMT is copied into the RMT to recover the correct register mappings. The branch mask field, which is part of every instruction packet dispatched to the backend, is used for selective flushing of instructions in the backend of the processor. The fetch unit starts fetching instructions from the correct program counter, as calculated by the execution unit.

3.3.2 Load violation recovery

As shown in Figure 3-2, the load and store instructions are dispatched into separate structures called the Load queue and the Store queue, respectively. These structures serve the purpose of memory disambiguation to resolve dependencies among load and store instructions. Load instructions are issued out of order, as soon as their addresses are known, irrespective of whether prior stores have obtained their addresses. When the load disambiguation logic detects an issued load instruction whose address matches with that of at least one prior store instruction, a load violation bit is set in the corresponding location in the Active List. When the violating load instruction reaches the head of the Active List, a load violation exception is raised, which results in the complete flushing of all pipeline registers in the processor, the Active List, the Issue Queue payload and CAM, and the load and store queues. The state of
RMT is restored by copying the architectural mappings of registers from AMT. The speculative free list is reset by rolling back its head pointer to equal the tail pointer, and the fetch unit starts fetching instructions starting from the violating load instruction.
Chapter 4 : Fault Vulnerability of Register Tags in a Superscalar Processor

Chapter 3 discussed the flow of register tags of an instruction through the various pipeline stages of a superscalar processor. This chapter dissects each pipeline stage for possible vulnerabilities in the combinational logic within, that would lead to a faulty register tag to be propagated and possibly used for the execution of an instruction.

4.1 Vulnerabilities in the Register Rename Stage

The rename stage of a superscalar processor has the functionality of renaming architectural or logical registers in the instruction word to physical registers, in order to eliminate false dependencies among instructions in a program. The MIPS ISA supports 32 logical registers that can be mapped to 128 physical registers, in the RTL implementation that is used for the current study. Figure 4-1 shows the main blocks of combinational logic along with the structures present in the rename stage.

The main combinational logic blocks constituting the register rename stage are:

1) Destination renaming logic to rename logical destinations of instructions, using registers popped out of the Speculative Free List.

2) Logic to take into account the output dependencies among instructions being renamed concurrently and update the Rename Map Table (RMT) accordingly.

3) Source register renaming logic to read out the mappings of source registers of instructions from the RMT.
4) Logic to check dependencies between source and destination registers being renamed concurrently.

5) Logic to copy RMT to one of the Shadow Map Tables (SMT) in case of checkpointing and to restore RMT from the corresponding SMT, during branch recovery.

The figure also shows the various possible locations that a fault might manifest and cause a faulty register number to be latched into the Rename-Dispatch pipeline register. Table 4-1 tabulates each fault along with its possible manifestation in the microarchitectural state of the processor.
Table 4-1 Effect of fault manifestations in Register Rename stage

<table>
<thead>
<tr>
<th>Fault number in Figure 4-1</th>
<th>Effect of fault</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Faulty physical destination being latched into Rename-Dispatch pipeline register</td>
</tr>
<tr>
<td>2</td>
<td>Faulty mapping being written into RMT</td>
</tr>
<tr>
<td>3</td>
<td>Faulty mapping in RMT</td>
</tr>
<tr>
<td>4</td>
<td>Faulty mapping being written into RMT</td>
</tr>
<tr>
<td>5</td>
<td>Faulty physical source being latched into Rename-Dispatch pipeline register</td>
</tr>
<tr>
<td>6</td>
<td>May manifest as faulty mapping in RMT</td>
</tr>
<tr>
<td>7</td>
<td>Faulty physical source being latched into Rename-Dispatch pipeline register</td>
</tr>
<tr>
<td>8</td>
<td>Faulty physical destination register or logical destination register or either of the physical source registers in Rename-Dispatch pipeline register</td>
</tr>
</tbody>
</table>

It is to be noted that even if a fault manifests in the microarchitectural state, it does not necessarily corrupt the architectural state. For example, if a fault manifests in one of the SMTs and the SMT is not used in the future for branch recovery, then the fault is masked. A detailed analysis of fault masking is carried out in Chapter 6.

4.2 Vulnerabilities in the Dispatch Stage

The Dispatch stage is concerned with routing the requisite information of an instruction to the Active List and the Issue Queue, as shown in Figure 3-2. Since no combinational logic exists as such, faults can manifest in the flip-flops of pipeline registers.
4.3 Vulnerabilities in the Issue Stage

A microarchitectural view of the Issue stage is shown in Figure 4-2, along with the various combinational logic blocks that could be affected by a fault. The issue stage comprises of the following structures –

1) the Issue queue (IQ) payload RAM holding information pertaining to each instruction that is yet to be granted
2) the source register CAM holding the source registers for each instruction in the IQ.

The combinational logic blocks in the Issue stage are:

1) Wakeup logic to compare the incoming destination tags with the source registers of each instruction in the IQ
2) Select logic to pick four ready instructions to be granted in the following cycle

Table 4-2 shows the effects of faults on the structures and combinational logic blocks holding and using register tags of instructions in the Issue stage.

<table>
<thead>
<tr>
<th>Fault number in Figure 4-2</th>
<th>Effect of fault</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Pre-mature wake up / consumption of wrong source data leading to incorrect execution result</td>
</tr>
<tr>
<td>2</td>
<td>Faulty Ready-bit array leading to pre-mature wake up or deadlock</td>
</tr>
<tr>
<td>3</td>
<td>Faulty logical or physical destination register of instruction</td>
</tr>
<tr>
<td>4</td>
<td>Pre-mature wake up or deadlock</td>
</tr>
<tr>
<td>5</td>
<td>Pre-mature wake up / deadlock</td>
</tr>
<tr>
<td>6</td>
<td>Faulty physical destination register or logical destination register or either of</td>
</tr>
</tbody>
</table>
4.4 Vulnerabilities in the Register-Read Stage

The register-read stage mainly comprises of reading out data from the physical register file. It also houses the RSR shift register that is used to wake up the consumers of a complex functional unit instruction, as explained in Chapter 3. A fault on the RSR shift register causes a wrong destination tag to be broadcast to wake-up consumers in the issue stage, and also causes the setting of the ready bit corresponding to the faulty destination register. If a fault occurs in the address decoding logic of physical register file, incorrect data is read and
latched into the Register read-Execute pipeline register. A fault may also occur in the pipeline register itself, resulting in a faulty source register being used by the execution unit, or a faulty physical destination register being used by the writeback stage.

4.5 Vulnerabilities in the Execute Stage

As mentioned in Chapter 3, physical source registers of an instruction are used in the Execute stage to grab the data available on the bypass bus. Incorrect physical source registers entering the execution unit may result in incorrect data (either due to incorrect match or incorrect mismatch between source registers and bypass tags) being used for execution, which in turn results in an incorrect outcome. The physical destination register tag of the instruction, though not used in the Execute stage, is also vulnerable since it is part of the instruction packet that is latched into the Execute-Writeback pipeline register.

4.6 Vulnerabilities in the Writeback Stage

The Writeback stage is involved with writing the execution result into the physical register file. A fault that occurs on the address decoding logic of the physical register file results in the updating of an incorrect location, which has the same effect as storing faulty data in the physical register file.

4.7 Vulnerabilities in the Retire Stage

The retire stage logically houses the Active List (AL) and the Architecture Map Table (AMT) SRAMs. Active List is a FIFO structure that stores instruction packets dispatched by the dispatch unit in program order, until the instructions are ready to be committed. An
instruction is considered ready for commit when all prior instructions have committed and the instruction has completed execution without any hardware exceptions. The relevant vulnerable fields of each instruction packet stored in AL SRAM are the logical and physical destination tags, which are used during instruction commit.

The AMT, as discussed in Chapter 3, contains the architectural mappings of logical registers to physical registers. Faults in the AMT are the most catastrophic since the AMT is part of the architectural state of the program visible to the higher abstraction layer. A fault in the AMT is propagated to the Speculative Free List (SFL) when a register mapping is freed from the AMT and pushed into the SFL. Additionally, any hardware exceptions, such as load violations, use the AMT to restore the state of the Rename Map Table (RMT) to continue program execution from the offending instruction. Hence, a faulty AMT, more often than not, would result in an erroneous program outcome.
Chapter 5: Proposed Microarchitecture-level Fault Check Regimen for a Superscalar Processor

Chapter 3 provided a detailed description of the flow of register tags throughout a superscalar processor. Chapter 4 delved into the possible vulnerabilities in the various pipeline stages that could corrupt register tags. This chapter proposes a series of microarchitecture-level techniques that can be used to detect faults in any of the structures and pipeline flip-flops that hold register tags. The proposed techniques are based on microarchitectural invariants that must hold true irrespective of the program being executed. These techniques exploit the knowledge of underlying microarchitecture and use existing hardware to detect faults in a manner that is non-intrusive and does not affect normal program execution. A suite of such techniques forms a comprehensive fault check regimen capable of providing an end-to-end coverage of faults affecting register tags.

This chapter is divided into various sections, each of which is dedicated to a fault detection technique. Each section describes the idea behind the fault check, its RTL implementation, and its expected coverage. Some techniques that have been proposed by [8] and [9], are explored here from an RTL perspective.

5.1 Register Re-renaming Techniques

The process of register renaming forms the heart of a processor capable of out-of-order execution. The register renaming process eliminates anti and output dependencies among
instructions, thereby facilitating their out-of-order execution. The proposed technique of register re-re-naming involves redundantly renaming instructions during their retirement. This technique exploits the redundancy between the rename map table (RMT) and the architectural map table (AMT) with respect to time. Clearly, the concept of register renaming requires the RMT to be ahead in time (hence, speculative) with respect to the AMT. However, since instructions are renamed as well as retired in program order, the same state of register mappings must be visible to an instruction during its renaming and retirement stages. This is the microarchitectural property of an out-of-order processor that is exploited by the technique of register re-re-naming. This conceptually simple technique of re-re-naming an instruction in its retirement stage provides a broad coverage for detecting faults in physical source and destination registers of an instruction, covering pipeline stages all through from the register rename stage up to the retirement stage.

5.1.1 Source register re-renaming

*Concept:*

In the register rename stage, as described in Chapter 3, an instruction’s logical source registers are used to read out the corresponding mappings from the RMT. Additionally, the dependency check logic also checks for dependencies among the source registers and destination registers of prior instructions, and renames the sources to the latest mappings. In source re-re-naming technique, this exact same process is repeated just before the retirement of an instruction in the commit stage, the only difference being, the physical register mappings of the logical sources are read out from the AMT. A redundant dependency check
logic in the commit stage takes care of dependencies among source registers and destination registers of prior instructions committing in the same cycle. These new physical register mappings determined in the commit stage are compared with the actual physical registers that the instruction used for execution, and they are asserted to be equal. An inequality will result if the instruction had consumed incorrect source operands for execution, either due to faulty renaming logic in the register rename stage or due to a fault in any of the pipeline latches between the rename stage and the execution stage.

**Implementation:**

For the source renaming process to be carried out at the commit stage, the logical source registers of every instruction need to be preserved until the instruction retires. In the baseline implementation, the logical source registers are discarded after the register-rename stage in the front-end. However, in the current modified implementation, logical source registers are part of the instruction packet that flows down the pipeline and is dispatched to the Active List. This adds 12 additional bits (6 for each logical source) to the Rename-Dispatch pipeline register as well as to the payload in the Active List.

For holding the actual physical source registers that were used by the instruction for execution until the instruction retires, an Auxiliary Active List (AxAL) is introduced. This structure is a physical extension to the already existing Control Active List, used to store the control bits of every instruction. After the completion of execution of every instruction, its physical source registers are written to the corresponding entry in the AxAL. When an
instruction is ready to retire, its physical source registers are read out from the AxAL, and sent to the retirement module that also houses the re-renaming logic.

The re-renaming logic in the commit stage requires eight read ports on the AMT SRAM to read out the mappings of two source registers of up to four instructions that can commit each cycle. This is in addition to the already existing four read ports, to read out the physical registers that are to be freed. Instead of using an SRAM with twelve read ports, the AMT is duplicated to form an Auxiliary AMT, owing to a smaller area overhead due to duplication. The Auxiliary AMT (AxAMT) SRAM contains eight read ports and four write ports. The read ports are used to read out source register mappings of the retiring instructions. The write ports are used to update the AxAMT at exactly the same time as the actual AMT and ensure that its register mappings are the latest.

**Fault Coverage:**

The purpose of the source register re-renaming technique is to detect faults affecting physical source register tags throughout the entire pipeline, beginning with the source renaming logic in the Register rename stage all the way up to the Execute stage. Figure 5-1 depicts the use of physical source register tags in each pipeline stage and the coverage of the source re-renaming technique.
Figure 5-1 (a) Implementation of source re-renaming technique, and (b) Fault coverage of source re-renaming technique

In consultation with Table 4-1 that tabulates the manifestation of faults in the register rename stage, and Figure 5-1, it can be inferred that the source re-renaming technique detects the following faults:
1) Fault in the address decoder logic while reading out mappings from the RMT: A faulty address reads out an incorrect source mapping from the RMT that is detected by source re-renaming during commit.

2) Fault in dependency check logic: A fault in the logic that determines dependencies among source registers and destination registers of instructions being renamed in the same cycle, might rename the source to an incorrect physical register.

3) Faults affecting physical source registers in all intermediate pipeline latches: Source re-renaming detects faults in all intermediate pipeline flip-flops that hold physical register tags until they are last used in the Execute stage. The pipeline registers covered are Rename-Dispatch, Dispatch-Issue, Issue-Register read, Register read-Execute.

4) Fault in the Rename Map Table (RMT): A fault in the RMT results in the logical source register being renamed to a faulty physical register. This is detected by the source re-renaming logic during instruction commit, due to the mismatch with the mapping read out from AxAMT.

5) Fault in the Shadow Map Table (SMT): A fault in any of the SMTs may manifest as a fault in the RMT if the faulty SMT is used to recover the RMT due to a branch misprediction. The scenario is then similar to that of a faulty RMT.

6) Fault in the Issue Queue CAM: The IQ CAM, as explained in Chapter 3, stores the physical source registers of all instructions yet to be issued. Any fault in the IQ CAM results in the instruction consuming data from an incorrect physical register. Since the
faulty physical source register is retained until commit, the fault is detected during re-renaming.

7) Fault in the AMT: A faulty mapping in the AMT may be propagated to the RMT due to a load violation recovery. However, since the AxAMT is not faulty, this fault is caught by source re-renaming.

8) Fault in the logical destination register tag: The logical destination of an instruction is used to update the corresponding entry in both the AMTs with the latest physical register mapping, during instruction retirement. A fault affecting the logical destination of an instruction during the course of its execution will result in the updating of the faulty entry in both the AMTs, thus resulting in two incorrect mappings in the AMTs. However, as long as the fault does not propagate to the RMT, it is detected by source re-renaming.

Thus, the source re-renaming technique provides a comprehensive end-to-end coverage and detects faults in not only the pipeline flip-flops holding source registers, but also important structures like the RMT and the Issue queue CAM. An important advantage of source re-renaming is that it also detects faults in the overall data flow, since it takes into account the instruction context and its dependencies with prior instructions. Moreover, it is non-intrusive to normal program execution since it is carried out in parallel with normal operation.
5.1.2 Destination register re-renaming

*Concept:*

As explained in Chapter 3, the logical destination of every instruction is renamed with a physical register that is popped by the Speculative Free List (SFL). The idea behind destination re-renaming is to maintain an Architectural Free List (AFL) in the commit stage that temporally (with respect to time) lags behind the SFL. During the commit of an instruction, the destination re-renaming logic pops a register from the AFL and asserts that it matches the destination register of the retiring instruction. This holds true because the SFL and the AFL, though staggered in time, are updated in exactly the same order of physical registers i.e. the register read out from the AMT is added to both the SFL and the AFL in the same clock cycle.

*Implementation:*

As depicted in Figure 5-2a, the physical destination of every instruction flows down both paths 1 and 2. The one that is passed down path 1 is used to update the AMT during instruction commit. However, the one in path 2 is used to write into the physical register file during writeback. Since a fault could occur anywhere in path 2 and corrupt the physical destination register the data is actually written to, an assertion check is required to verify its correctness. The original RTL implementation, where the physical destination register is discarded after writeback stage, is modified to hold the physical destination register in the Auxiliary Active List (AxAL), along with the physical source registers, as described in section 5.1.1.
In the retirement stage, two comparisons are performed among three physical destination tags for every instruction-

1) comparison between physical destination tag from path1 and physical destination tag from path2
2) comparison between physical destination from path1 and physical register popped from the Architectural Free List.

If either of these comparisons results in a mismatch, a physical destination fault assertion is raised.

*Fault coverage:*

The destination re-renaming technique provides coverage against faults that affect physical destination tags throughout the processor, as depicted by Figure 5-2b.
Figure 5-2 (a) Implementation of destination re-renaming, and (b) Fault coverage of destination re-renaming

Following is the list of structures that are protected by destination re-renaming:

1) Fault affecting physical destination tags in pipeline registers: Destination re-renaming detects faults in physical destination tags stored in all the intermediate pipeline registers appearing in path 2. Path 2 comprises of the Issue-Register read, Register read-Execute and Execute-Writeback pipeline registers. A fault on the physical
destination register in any of these pipeline flip-flops is detected, albeit only when the instruction retires. This shortcoming of destination re-renaming is addressed with an additional fault detection technique, called Ready-bit status check II, which is discussed later.

2) Fault in the Active List RAM: The Active List stores physical destination register of every instruction as part of its instruction payload, which is later used to update the Architecture Map Table during instruction retirement.

3) Fault in the Issue queue payload RAM: The Issue queue stores physical destination register as one of the components of its payload, which is passed on to the later pipeline stages when the instruction is issued. Thus, a fault in the IQ payload is equivalent to a fault in any of the pipeline registers in path 2.

4) Fault in the Speculative Free List: A major structure that is protected by destination re-renaming is the SFL, since an alternate Architectural Free List is maintained in the commit stage to assert the correctness of SFL.

5) Fault in the logic that checks output dependency in the register rename stage: Due to the replication of the logic to rename destination registers during commit, faults in the logic that renames destination registers in the register rename stage are automatically caught during re-renaming.

The destination re-renaming technique proves very effective in providing an end-to-end protection to the physical destination tags, from the rename stage up to the retirement stage. However, faults affecting the destinations of instructions in mispredicted paths are not
detected by the re-renaming technique, since such instructions never reach the retirement stage. The Ready bit status check II, which is discussed in Section 5.3, detects most of that faults that evade re-renaming.

5.2 Previous Mapping Check

The previous mapping assertion check augments the register re-renaming techniques discussed above, by covering faults in certain structures and scenarios not covered by the re-renaming techniques. This technique was introduced in [9] as one of the techniques for register name authentication.

Concept:

The previous mapping check exploits the conformity between the Rename Map Table (RMT) and the Architecture Map Table (AMT) during fault-free operation. While, the AMT maps every logical register to its latest committed version, the RMT represents the latest future version of every logical register. Since, renaming and retirement of an instruction happen in program order, the state of the RMT as seen by an instruction during register renaming should exactly match the state of the AMT the stale mapping of its logical destination in the RMT during renaming must be the same as the stale mapping in the AMT during retirement. The previous mapping check asserts this microarchitectural property for every retiring instruction. This is better understood with an example.

Let I1 be an instruction that writes to logical register L1. Following are the states of the processor during the renaming and retirement stages of the instruction:
1) Renaming of I1

II: \( D_{l1} \leftarrow S_{l1}, S_{l2} \) is renamed to \( D_{p100} \leftarrow S_{p1}, S_{p2} \)

<table>
<thead>
<tr>
<th>RMT (before renaming II)</th>
<th>RMT (after renaming II)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logical</td>
<td>Physical</td>
</tr>
<tr>
<td>( D_{l1} )</td>
<td>( D_{p50} )</td>
</tr>
<tr>
<td>( S_{l1} )</td>
<td>( S_{p1} )</td>
</tr>
<tr>
<td>( S_{l2} )</td>
<td>( S_{p2} )</td>
</tr>
</tbody>
</table>

Thus, in RMT, previous mapping of \( D_{l1} \) is \( D_{p50} \) and its new mapping is \( D_{p100} \). The previous mapping of the destination register is retained until instruction retirement.

2) Retirement of I1

<table>
<thead>
<tr>
<th>AMT (before update by II)</th>
<th>AMT (after update by II)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logical</td>
<td>Physical</td>
</tr>
<tr>
<td>( D_{l1} )</td>
<td>( D_{p50} )</td>
</tr>
<tr>
<td>( S_{l1} )</td>
<td>( S_{p1} )</td>
</tr>
<tr>
<td>( S_{l2} )</td>
<td>( S_{p2} )</td>
</tr>
</tbody>
</table>

When I1 reaches the Retirement stage, the AMT mapping of its logical destination \( D_{l1} \) is read out and asserted that it matches the previous mapping of the RMT recorded earlier in the Register renaming stage. The AMT is updated with the new mapping only after the stale mapping has been read out.

The above tables show the fault-free states of the RMT and the AMT during renaming and retirement stages of instruction I1, respectively.
While the correctness of the physical register tag used to update the AMT is verified by the destination re-renaming logic (discussed in Section 5.1), this assertion technique verifies the AMT entry that is updated with the new physical register tag.

**Implementation:**

The implementation of previous mapping check requires reading out the mappings from the RMT before they are updated with the new mappings. The stale mapping is appended to the instruction packet that is dispatched to the Active List, and hence requires an additional 7 bits in each entry of the Active List. During an instruction’s retirement, the mapping in the AMT corresponding to its logical destination is read out before updating it with the new physical register. The old mapping of the AMT is compared with the stale RMT mapping held in the Active List, and they are asserted to be equal. The process of reading out the old mapping from the AMT is part of the existing implementation, since it is required for freeing the old physical registers.

**Fault Coverage:**

As mentioned above, the previous mapping check complements the destination re-renaming check in detecting faults in destination registers after renaming since, it detects faults in the logical destination registers, while destination re-renaming detects faults in physical destination registers. Following is the independent fault coverage provided by the previous mapping check:
1) Faults in pipeline registers storing physical destination tags: The previous mapping check is capable of detecting faults in the physical destination register that is used to update the AMT, although the detection occurs only when the next producer of the same logical register retires. Suppose a fault occurred in one of the pipeline registers holding the renamed instruction packet of I1, thereby modifying its physical destination register from P2 to P50. This results in updating the AMT with the faulty physical register P50. A future instruction I2 with the same logical destination as I1, during its renaming, gets P2 as its previous mapping from the RMT. During the retirement of I2, the previous mapping from RMT (P2) mismatches with the mapping read out from the AMT (P50), thereby detecting the fault.

2) Faults in the Rename Map Table: When an instruction records a faulty RMT mapping as the previous mapping of its logical destination during register renaming, the fault is detected during the instruction’s retirement, due to the mismatch with the old mapping read out of the AMT.

3) Faults in the Architectural Map Table: A faulty AMT entry is detected due to its mismatch with the previous RMT mapping of that logical register, during the retirement of the logical register’s producer. This prevents the fault from being propagated to the Speculative Free List when the faulty register is added to the SFL. It should be noted that, if a faulty AMT is used for RMT recovery due to an exception or a load violation, the RMT, though faulty, becomes consistent with the AMT. Hence, in this case, the fault will never be detected by the previous mapping check.
4) Faults in the Active List payload: As mentioned earlier, the logical destination of every instruction is stored in the Active List until retirement. A faulty logical destination is detected due to the incorrect mapping that is read out from the AMT and the resulting mismatch with the previous RMT mapping. This prevents the updating of an incorrect AMT entry, thereby preventing corruption of the architectural state of the processor. A faulty physical destination, stored as part of the Active List payload, is detected similar to case 1 above (Faults in pipeline registers storing physical destination tags). Since previous RMT mapping is also a part of the Active List payload, a faulty previous mapping results in a false firing of the assertion check.

5.3 Register Readiness Checks

While the checks discussed so far assert the correctness of source and destination register tags of an instruction, the register readiness checks assert the timing involved in reading from the physical source registers and writing to the physical destination registers. Every physical register in the microprocessor that is used for program execution has a ready bit associated with it that indicates whether the register is ready for consumption by the instruction. The register readiness checks assert the status of the ready bit at two different instances during the execution of every instruction. It is to be noted that these ready bits are different from those that are present in the issue stage, which are used by the selection logic to issue instructions whose source registers are ready.
The ready bit status checks described here are specific to the current baseline implementation, though the concept is general enough to be incorporated into any implementation. In the current RTL implementation, the ready bit of a physical register is cleared when the register is used to rename the destination of an instruction. The setting of the ready bit happens at different instances, depending on the instruction. In case of instructions whose latencies are known, it is set one clock cycle prior to the completion of instruction execution. In case of load instructions, the ready bit of the destination register is set only when the data is available. The ready bit of a register remains set even when it is displaced from the Architectural Map Table (AMT) and added to the Free List. It is cleared only when it is re-used by the rename logic for destination renaming.

5.3.1 Ready bit status check I

*Concept:*

The ready bit status check I (RBS I) basically detects a Read After Write (RAW) hazard by asserting that the ready bits of the physical source registers of every instruction are set while their corresponding data are being read out from the physical register file in the Register read stage. This holds true for back-to-back issue of dependent instructions as well, because the ready bit of the destination register of the producer is set in the same cycle as the destination tag is broadcast to the Issue Queue CAM.
Implementation:

The implementation of RBS I does not require any major addition or modification to the current RTL implementation. The ready bits of each of the valid source registers of the instruction entering register-read stage are asserted to be set (i.e., 1). Figure 5-4 shows the Issue stage, the Register-read stage, and the RBS I assertion check.

Fault Coverage:

The RBS I check provides coverage against faults that result in premature issue of instructions in the Issue Queue, which might be due to:

1) fault in the source ready bits associated with each instruction in the IQ
2) fault in the wakeup logic of Issue stage
3) fault in the destination tags that are broadcast to the Issue Queue
4) fault in the select logic of Issue stage

5.3.2 Ready bit status check II

Concept:

While RBS I asserted the status of ready bits of physical source registers, RBS II is intended to assert the ready bit status of the physical destination register while it is written to. This fault check is described in [9], where it is referred to as RNA2 or Writeback state check. The RBS II check detects the occurrence of write-after-write (WAW) and write-after-read (WAR) data hazards, by asserting that the ready bit of the destination physical register of an instruction is clear before setting it. The reasoning behind this assertion is that during a major
portion of the lifetime of a physical register, its ready bit is set, and any fault affecting the physical destination register of an instruction is more likely to cause a conflict with another register whose ready bit is already set.

**Implementation:**

The originally proposed idea in [9] performs this assertion check in the Writeback stage, because in the implementation assumed therein, the setting of ready bit is postponed until the instruction completes execution. However, in the current implementation and most other implementations, the ready bit is set before the instruction completes execution (except for load instructions). Hence, this assertion check is incorporated before the Writeback stage, just before setting the ready bit of the physical destination register, which generally takes place in the Register read stage. The assertion check for load instructions is performed only after the data is available, because that is when the ready bit of its destination register is set.

Figure 5-4 shows a high-level block diagram of the Issue stage and the Register read stage, along with the timing involved in the process of ready bit status checks. As shown, the RBS II assertion check happens in parallel with the reading of physical register file, just before setting the ready bits of the destination registers.

**Fault coverage:**

The RBS II check is intended to check faults in physical destination registers of instructions, to ensure that data is written to the correct register. Since this check makes sure that the ready
bit is not already set before setting it, RBS II covers a major portion of the lifetime of a physical register.

1) Firstly, it ensures that an instruction does not write to a register that is the destination of an already executed instruction. The fault could either be in the destination of the instruction that has already finished execution or it could be in the destination of the instruction that is trying to write to the same physical register.

2) Secondly, it ensures that no instruction tries to write to a physical register that is currently ‘free’ i.e., a register in the Speculative Free List. This is because, in the current implementation the ready bit of a physical register also acts as its free bit, depending on state of the register. If a physical register is part of the architectural stage or is the destination of an in-flight instruction, then the ready bit indicates whether the register is ready for consumption. On the other hand, when a register is displaced from the AMT and added to the Free List, its ready bit remains set, since, as mentioned earlier, the ready bit is cleared only when the register is used to rename the destination of an instruction. This subtle dual-purpose role of the ready bit is used to assert that an instruction does not write to a free register.

If the baseline implementation has a ‘free’ bit as well associated with every register, an alternate implementation of RBS II would be to assert that neither, the ready bit and the free bit, is already set, before setting the ready bit.
Thus, the RBS II is capable of detecting faults in physical destination registers stored in the following structures:

1) Pipeline registers: Faults affecting the physical destination tags in the Rename-Dispatch, Dispatch-Issue, Issue-Register read pipeline registers may get detected by RBS II.

2) Issue Queue payload: A fault in the destination register field of the IQ payload is equivalent to that in any of the above pipeline registers.

3) Speculative Free List: RBS II is capable of detecting faults in the SFL when the faulty register conflicts with the non-faulty version of the same register.

4) Architectural Free List: When a faulty register is displaced from the AMT during the freeing process, the fault is transferred to the Free List. This scenario is similar to that of a fault affecting the Free List discussed above.

It is to be noted that RBS II detects faults only in case a conflict occurs between the correct instance and the faulty instance of a physical register. Destination re-renaming technique, discussed in Section 5.1.2, provides a much more comprehensive protection to destination register tags. However, RBS II fault check is necessary to detect faults in certain corner cases when the fault evades destination re-renaming at commit.

Figure 5-3 shows two corner cases where the fault evades destination re-renaming, but is capable of being detected by RBS II.
Corner cases that evade destination re-renaming but detected by RBS II

In case 1 of Figure 5-3, the fault affects the destination of an instruction, I4, belonging to a mispredicted path. I4 is issued and corrupts the contents of P1, thereby resulting in I2 consuming an incorrect data value. However, since I4 never reaches the retirement stage, destination re-renaming will not be able to detect the fault. Case 2 is similar to case 1, the difference being, the faulty register is consumed a branch instruction, thereby changing the path of execution. In this case, even though the fault occurred on an instruction in the right path, the fault changes the course of execution, resulting in the flushing of the faulty instruction.
From the above discussion, it is clear that RBS II protects faults in the destination register fields of all pipeline registers that exist prior to the register read stage, which is when the ready bit of the physical destination register is set. However, the pipeline registers following the register-read stage are still vulnerable to faults that can change the course of a program, and evade destination re-renaming fault check. One solution to deal with such scenarios is to maintain a secondary ready-bit array that is logically updated in the writeback stage, when the physical register is actually written to. It could then be asserted that the secondary ready bit of the destination register is not set before writing into the register. The current work does not implement this fault check.
Figure 5-4 Issue and Register Read pipeline stages with RBS I and RBS II fault checks
5.4 Watchdog Timer

Watchdog timer is a commonly used tool to detect deadlocks in the state of a system. In case of microprocessors, a watchdog timer can be effectively used to keep track of instruction retirement, and detect deadlocks if no instruction commits for a preset number of clock cycles. In a typical superscalar processor, the average time an instruction resides in the Active List is in the order of tens of clock cycles. Thus, a 10-bit watchdog timer is sufficient to detect deadlocks preventing instruction retirement for thousand cycles. The timer is set to its maximum value every clock cycle when at least one instruction commits. If no instruction commits in a given clock cycle, the counter is decremented by 1. If the counter value reaches 0, a fault signal is raised to indicate a deadlock.

Fault coverage:

From the perspective of faults affecting physical registers throughout the processor, the watchdog timer detects faults in the following scenarios:

1) Deadlock between a producer and its consumer: A fault on the source register of an instruction may change it to the destination of its consumer, in which case, the consumer and the producer are both waiting for each other to execute, causing a deadlock.

2) Deadlock due to phantom source register: A fault on the source register of an instruction after it has been renamed may cause the instruction to wait on a faulty destination tag which may never arrive, causing the instruction to be never issued.
3) Fault on the source ready bit in the issue stage: A fault on the source ready bit of an instruction may flip the bit to zero, thereby causing the instruction to wait on a destination tag that has already been broadcast and is ready to be consumed.

Such corner cases are not detected by any of the fault detection techniques discussed earlier, and hence necessitate the inclusion of the watchdog timer as part of the fault check regimen.

5.5 A Unified Fault Check Regimen

The fault detection techniques discussed in this chapter form a unified fault check regimen to detect faults in register tags throughout a superscalar processor. The regimen provides fault coverage of almost 100% for the targeted structures, as will be shown in Chapter 7.

Figure 5-5 shows the structural additions to a superscalar processor in order to incorporate the techniques discussed. Table 5-1 and Table 5-2 summarize the various targeted structures and the techniques that protect them. It should be noted that the current work only targets faults affecting register tags anywhere in the processor. Although no technique’s coverage is comprehensive enough to detect all faults, the collective coverage of the regimen spans across all pipeline stages and targeted structures.
Table 5-1 Fault coverage of pipeline flip-flops by the fault check regimen

<table>
<thead>
<tr>
<th></th>
<th>Source re-renaming</th>
<th>Destination re-renaming</th>
<th>Previous RMT Mapping</th>
<th>Ready bit status check I</th>
<th>Ready bit status check II</th>
<th>Watchdog</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rename-Dispatch</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Issue-Register-read</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Register-read-Execute</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Execute-Writeback</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 5-2 Fault coverage of RAMs and CAMs by the fault check regimen

<table>
<thead>
<tr>
<th></th>
<th>Source re-renaming</th>
<th>Destination re-renaming</th>
<th>Previous RMT Mapping</th>
<th>Ready bit status check I</th>
<th>Ready bit status check II</th>
<th>Watchdog</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rename Map Table</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Architectural Map Table</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Speculative Free List</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Issue queue payload RAM</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Issue queue CAM</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Active List</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Figure 5-5 Register tag flow after structural additions (in bold outline) as part of incorporating the proposed fault check regimen
Chapter 6: Fault Injection Methodology

This chapter describes the methodology used in the current work to inject faults into the Verilog RTL model of a superscalar processor. The fault model used in the current work is a single bit flip, either in a state element, like a pipeline flip-flop, or an array based structure, such as the Rename Map Table. Since the current work mostly concentrates on detecting faults on register tags throughout a processor, the number of bits representing register tags that are potential targets for fault injection can be determined using Table 6-1 to be around 5000. Considering the number of instances in time when each bit could be flipped, the total number of possibilities for injecting a fault would be extremely high. This necessitates sampling in both the space domain and the time domain for injecting faults and at the same time, obtain reasonably significant results.

For sampling in the space domain, every set of bits representing a register tag is assigned an identification number. For example, the Rename Map Table would be assigned 32 sequential numbers, since there are 32 entries in the RMT holding register tags. Similar number assignment is done to every register tag field in every pipeline register and structure holding register tags. The fields that are added as part of the unified fault-check regimen are also assigned numbers, since they are equally susceptible to faults. Thus, the granularity for fault injection is an entire register tag field rather than each bit of every register tag field. Bit-level granularity is provided only for fault injection into the ready bit array in the Issue module. Sampling in the time domain is done by choosing an arbitrary cycle between 50,000 and
100,000 clock cycles after the start of simulation, for fault injection. This allows for any necessary warm-up required for the processor.

### Table 6-1 Fault number assignment to various structures

<table>
<thead>
<tr>
<th>Structure</th>
<th>Number of fields with register information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rename - Dispatch pipeline latch</td>
<td>Logical sources(^1) (8), Physical sources (8), Logical destination (4), Physical destination (4), Previous RMT mapping(^1) (4)</td>
</tr>
<tr>
<td>Issue - Register read pipeline latch</td>
<td>Physical sources (8), Physical destination (4)</td>
</tr>
<tr>
<td>Register read - Execute pipeline latch</td>
<td>Physical sources (8), Physical destination (4)</td>
</tr>
<tr>
<td>Execute - Writeback pipeline latch</td>
<td>Physical sources(^1) (8), Physical destination (4)</td>
</tr>
<tr>
<td>Rename Map Table</td>
<td>32</td>
</tr>
<tr>
<td>Shadow Map Tables</td>
<td>4*32</td>
</tr>
<tr>
<td>Speculative Free List</td>
<td>96</td>
</tr>
<tr>
<td>Issue Queue Payload</td>
<td>Physical destination (32)</td>
</tr>
<tr>
<td>Issue Queue CAM</td>
<td>Physical sources (64)</td>
</tr>
<tr>
<td>Issue Queue Ready bit array</td>
<td>64</td>
</tr>
<tr>
<td>Active List</td>
<td>Logical sources(^1) (128<em>8), Logical destination (128</em>4), Physical destination (128<em>4), Previous RMT mapping(^1) (128</em>4)</td>
</tr>
<tr>
<td>Auxiliary Active List(^1)</td>
<td>Physical sources (128<em>8), Physical destination (128</em>4)</td>
</tr>
<tr>
<td>Architecture Map Table</td>
<td>32</td>
</tr>
<tr>
<td>Auxiliary AMT(^1)</td>
<td>32</td>
</tr>
</tbody>
</table>

\(^1\) Fields/structures added as part of the fault check regimen

### 6.1 Fault Injection Environment

A hierarchical fault injection environment, shown in Figure 6-1, is used for injecting faults into the RTL model of the superscalar processor. A fault injection experiment comprises of multiple trials (runs), where each trial’s input is characterized by a fault number and a fault
injection cycle, and each trial’s output is characterized by a finish code representing the outcome. The input parameters for each trial are randomly generated by a top-level perl script and passed on to the Verilog testbench, which is responsible for injecting the fault at the specified clock cycle. At the lowest level, fault injection infrastructure is integrated into the RTL model of the microprocessor, but excluded from synthesis. Decoding logic is incorporated into every relevant Verilog module to decode the fault number received from the testbench and inject a fault if the number lies in the range for the particular module. Fault injection involves flipping a pre-decided bit of the corresponding register tag field in the structure. If the structure happens to have multiple possible register tag entries (such as the Active List), a local random number generator is used to select the entry number to be injected with the fault. The faulty bit is pre-decided to be the most significant bit of the register tag field, in order to avoid explosion in the number of possible locations for fault injection, and simplify reconstruction of the original register tag for later analysis. Since, all the fault detection techniques discussed in the current work are independent of the location of the faulty bit in a given register tag field, this decision will not significantly affect the fault coverage results. The ‘Retry’ feedback path from the RTL model to the Verilog testbench is explained in Section 6.2.
Other than the decoding logic in each module, a vital addition to the RTL model is the incorporation of a mechanism to track the faulty register tag from the top level testbench until the retirement of the instruction using the faulty register tag. For this purpose, a sticky bit, henceforth referred to as the ‘faulty’ bit, is maintained for every register tag field in all array based structures holding register tags, such as the Rename Map Table, the Architectural Map Table, the Free List, the Issue Queue and the Active List. A similar bit is appended to every instruction packet in every pipeline register, to indicate if any of the register tag fields in the instruction packet is faulty. If the ‘faulty’ bit of a retiring instruction is set, the event,
referred to as a ‘faulty’ commit, is recorded by the testbench. The occurrence of ‘faulty’ commit is used for outcome analysis, as will be made clear in Section 6.2.

6.2 Analysis of Possible Outcomes

During the execution of each experimental trial, the Verilog testbench interfaces with a golden functional C++ simulator to check the correctness of every instruction that commits to the architectural state. As part of the functional correctness check, the instruction’s program counter, its logical destination and the result of instruction’s execution are cross-verified with the functional simulator. If any of the above fields do not match, then an architectural corruption is flagged and the Verilog testbench is notified of the same. The Verilog testbench, however, does not stop the simulation until one of the following conditions is satisfied –

1) the testbench records architectural corruption as well as the firing of one or more of the assertions

2) simulation reaches the end of observation period

Note that in the first scenario, the order of occurrence of the two events does not matter. As long as one or more of the assertions fire and architectural corruption occurs, the fault is considered to be detected.

The second scenario occurs when none or only one of the events in scenario 1 occurs. Due to the dynamic nature of pipeline registers, the Active List and the Free List, it was observed that an observation period of 10,000 is sufficient to observe the effect of a fault on these
structures. However, for experiments involving fault injection on the Rename Map Table and the Architectural Map Table, an observation period of 100,000 cycles is used.

At the end of every trial, the Verilog testbench passes a unique finish code to the top level script depending on the outcome of the simulation. The various possible outcomes of each simulation are discussed below:

1) *Neither architectural corruption nor ‘faulty’ commit by the end of observation period (masked)* – The fault is said to be ‘masked’ if it did not propagate to the microarchitectural state, either directly or indirectly. There are several reasons for microarchitectural masking, some of which may be:
   a) the chosen register field contained an invalid source or destination operand in the chosen cycle
   b) the chosen cycle happened to be a stall cycle, due to which the faulty entry in the pipeline register was overwritten by the correct entry in the following cycle
   c) the fault was injected into an invalid location, such as an entry in the Active List containing no valid instruction
   d) the fault was flushed due to a load violation or a branch misprediction, before it could cause corruption

Since the aim of the experiments is to determine the coverage of the proposed fault check regimen, it is desirable to have as few masked faults as possible. Hence, most of the above scenarios that lead to masking are taken care of by imposing localized
constraints in every module, using the information that is available at that point in time (for example, the presence of an invalid source operand is known at the time of fault injection). In such a scenario, the Verilog testbench waits for a small random number of clock cycles and attempts to re-inject a fault at the same location (depicted as ‘Retry’ in Figure 6-1). In spite of all the constraints at the module level, all trials that result in this outcome are those in which the fault is flushed due to branch misprediction or load violation. This is an unavoidable case of microarchitectural masking, since the branch outcome or the load outcome is not known at the time of fault injection. Note that, even if one or more assertions fire but no ‘faulty’ commit is observed, the fault is considered to be masked.

2) ‘Faulty commit’ with no architectural corruption by the end of observation period (benign) – This outcome occurs if the fault successfully manifested but its manifestation did not cause architectural corruption and hence termed ‘benign’. The successful manifestation of the fault is inferred from the occurrence of ‘faulty’ commit, implying that an instruction that used a faulty register tag committed to the architectural state without causing corruption.

3) Architectural corruption with firing of at least one assertion (detected) – This outcome occurs when the fault causes architectural state corruption, and is detected by one or more assertions. In this case, the simulation is not continued until the end of observation period, but terminated as soon as both these events have occurred. The Verilog testbench returns a unique finish code to the top level script for each of the
abbreviations that fires. It is to be noted here that, in case more than one assertion fires, the assertion that fired closest to the cycle of corruption is credited for detecting the fault.

4) **Architectural corruption with no firing of assertion by the end of observation period (undetected)** – This outcome is the most important of all, since it occurs when the fault goes undetected by any of the assertion checks and causes silent data corruption. The number of undetected faults is a measure of the effectiveness of the fault check regimen.

5) **Neither architectural corruption nor ‘faulty’ commit by the end of observation period, but fault still exists in the microarchitectural state (dormant)** – This is a special case of outcome 1, which occurs when faults are injected into the Architectural Map Table (AMT). This outcome results when the fault manifests in an AMT entry corresponding to a logical register that is not used by the program after the fault manifests. Thus, even though the fault is still part of the microarchitectural state, it does not cause corruption within the observation window. The fault is termed dormant since it still is capable of corruption in the future.
Chapter 7 : Fault Injection Results

This chapter discusses the observations and results of the fault injection experiments performed as outlined in Chapter 6. A series of experiments were performed, each focusing on a specific structure in the superscalar processor. Every experiment consisted of 500 trials, which was observed to be good enough to yield meaningful results. For structures containing multiple eligible fields for fault injection, the 500 trials were divided among them proportionally. For example, 500 trials were divided among the 32 entries in case of the Rename Map Table. Since experiments were performed on the RTL model incorporating the unified fault check regimen, the additional hardware was also subjected to fault injection. The reader is encouraged to consult Figure 5-5 to identify the various vulnerable structures containing register tags.

Each trial’s outcome, as discussed in Chapter 6, could be one of the following – Masked (Dormant, in case of AMT), Benign, Previous RMT Mapping assertion (Prev Map), Ready-bit status check I, Ready-bit status check II, Source re-renaming (Src RR), Destination re-renaming (Dest RR), Watchdog, or Undetected. The results are presented for five benchmarks of the SPEC CINT2000 suite – bzip, gap, gzip, mcf and vortex, and averaged over these benchmarks. Although, the coverage results, ideally, must not differ across benchmarks, the degree of fault masking is directly related to the branch prediction accuracy and the number of load violations, in the chosen simulation period. As shown in Table 7-1, mcf has the least number of load violations and branch mispredictions, while gzip has the
maximum number of load violations and branch mispredictions combined. The effect this has on fault coverage will be evident during the analysis of obtained results.

Table 7-1 Relevant benchmark measurements at the end of 100,000 clock cycles

<table>
<thead>
<tr>
<th></th>
<th>bzip</th>
<th>gap</th>
<th>gzip</th>
<th>mcf</th>
<th>vortex</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of branches</td>
<td>17506</td>
<td>8812</td>
<td>5425</td>
<td>22872</td>
<td>11665</td>
</tr>
<tr>
<td>Number of branch mispredictions</td>
<td>243</td>
<td>1997</td>
<td>2130</td>
<td>1</td>
<td>1645</td>
</tr>
<tr>
<td>Branch misprediction rate</td>
<td>1.384%</td>
<td>22.67%</td>
<td>39.26%</td>
<td>0.0043%</td>
<td>14.10%</td>
</tr>
<tr>
<td>Number of loads</td>
<td>25617</td>
<td>18416</td>
<td>15357</td>
<td>45746</td>
<td>24882</td>
</tr>
<tr>
<td>Number of load violations</td>
<td>793</td>
<td>769</td>
<td>914</td>
<td>0</td>
<td>1044</td>
</tr>
<tr>
<td>Number of committed instructions</td>
<td>130959</td>
<td>82282</td>
<td>67234</td>
<td>106739</td>
<td>79747</td>
</tr>
<tr>
<td>IPC</td>
<td>1.31</td>
<td>0.82</td>
<td>0.67</td>
<td>1.06</td>
<td>0.80</td>
</tr>
</tbody>
</table>

7.1 Rename-Dispatch Pipeline Register

The fields that are targeted in the pipeline register between the Rename stage and the Dispatch stage are the physical source register tags, physical destination register tags and the logical destination registers. Additionally, the fields added in the new implementation – the logical source registers and the previous RMT mapping – are also vulnerable to faults. Figure
7-1 shows the fault coverage for each of the fields in the Rename-Dispatch pipeline latch for various benchmarks.

As is evident from the figure, the number of masked faults is zero for *mcf* while they are on an average 65% of the injected faults for *gzip*. Each of the fields has a dominant fault check that detects all or most of the faults affecting the field. Most of the faults in the physical source register tags are detected by source re-renaming, while a small percentage of those are detected by the watchdog timer. Around 10% of physical source register tag faults are benign, indicating that faulty source register consumption did not cause corruption. This is mostly the case when the execution result of the instruction is zero, irrespective of the consumed source or when the instruction has an immediate source operand, in which case, a faulty source register field does not affect instruction outcome. Faults in the physical destination tag field are mostly detected by destination re-renaming. While destination re-renaming is capable of detecting all such faults, ready-bit status check II detects corruption in some cases, even before the instruction reaches commit stage. As expected, all non-masked faults in the logical destination field are detected by the previous mapping check. Faults affecting the logical source register and the previous mapping fields are benign and do not affect program outcome, since these fields were added as part of the implementation of fault check regimen. However, such faults do cause unnecessary triggering of fault check assertions.
7.2 Issue-Register Read Pipeline Register

The coverage statistics for the pipeline register between Issue stage and Register read stage are shown in Figure 7-2. The physical source register and the physical destination register fields are the targeted entities in this pipeline latch. The percentage of masked faults is slightly smaller compared to the rename-dispatch pipeline register. However, as expected, most of the faults in the physical source register fields are caught by the source re-renaming technique, while some of them are caught by the ready-bit status check I. Faults on the physical destination register fields are detected by the ready-bit status check II and the destination re-renaming check. As discussed in Chapter 5, either of them is not capable of providing complete coverage and hence both are necessary for obtaining 100% coverage.
Figure 7-2 Fault injection results for Issue-Register Read pipeline register

7.3 Register Read-Execute Pipeline Register

The physical source and destination register fields in the pipeline register between Register read stage and the Execute stage are the vulnerable entities targeted for fault detection. It can be observed from Figure 7-3 that all non-masked faults affecting the physical source register information are benign and never result in corruption. This is due to the limitation of the current RTL implementation since it does not support back-to-back execution of the producer and consumer instructions. Hence, the physical source registers are never used by an instruction after the Register read stage, which otherwise would have been used for reading out data off the bypass bus.

Another noticeable statistic in Figure 7-3 is the presence of undetected faults in the physical destination register. These undetected faults, as discussed in Chapter 5, are due to the
corruption caused by instructions in the wrong path. The fault occurs after the ready-bit status check II and evades the destination re-renaming check since, the instruction never reaches the commit stage. This exposes a loophole in the current fault check regimen, resulting in less than 100% coverage for faults in the Register read-execute pipeline latch.

![Figure 7-3 Fault injection results for Register Read-Execute pipeline register](image)

### 7.4 Execute-Writeback Pipeline Register

The statistics for the Execute-Writeback pipeline latch are very similar to those for the Register read-Execute stage, as shown in Figure 7-4. The physical source register tags, added as part of the fault check regimen, do not affect the program outcome, and hence their faults are benign. Most of the faults on physical destination register tags are detected by destination re-renaming, except for a small percentage of undetected faults. This scenario is same as the one that causes undetected corruptions in the Register read-Execute pipeline latch.
Figure 7-4 Fault injection results for Execute-Writeback pipeline register (starred fields ‘*’ were added as part of the unified fault check regimen)

7.5 Architectural Map Table

Figure 7-5 shows the coverage statistics for faults in the Architectural Map Table (AMT). Unlike all other structures, the AMT represents the committed state of the processor, and hence faults in the AMT are not masked by load violations and branch mispredictions. However, as discussed in Chapter 6, AMT is subject to faults that do not get flushed, but do not affect the program outcome either. Such faults are depicted as ‘dormant’ faults in Figure 7-5. To justify the presence of these dormant faults, Figure 7-6 shows the usage of logical registers by the *bzip* benchmark in the observation period and the distribution of register numbers chosen for fault injection. It can be noticed that if a fault occurs on a register that is never used through the end of observation period, it remains a dormant fault, neither causing
corruption nor firing any of the assertion checks. From Figure 7-5, it can be deduced that \textit{gap} benchmark has the maximum utilization of logical registers, owing to the least number of dormant faults in the observation period. This is consistent with the register usage statistics obtained separately from the \textit{gap} benchmark. It should be noted that the observation period for different benchmarks was chosen based on the 100,000 cycle interval (up to an absolute cycle count of 500,000) having the maximum register utilization.

![Figure 7-5 Fault injection results for Architectural Map Table](image)

\textbf{Figure 7-5 Fault injection results for Architectural Map Table}
Figure 7-6 Register usage and Fault distribution for the bzip benchmark

Faults occurring in non-dormant registers of the AMT are capable of propagating to the following other structures in the processor, before being detected –

1) Speculative Free List and the Architectural Free List – The fault is transferred to these structures when the faulty register is freed from the AMT. In this case, since
both the Free Lists are consistently faulty, destination re-renaming check would not be able to detect the fault.

2) Rename Map Table (RMT) – In case of a load violation, the AMT is copied onto the RMT, thereby \textit{propagating} the fault to the RMT. This scenario makes the RMT consistent with the faulty AMT, thus defeating the idea behind re-renaming.

Most of the faults that result in either of the above scenarios are successfully detected by the fault check regimen as is evident from a very small percentage of undetected faults in Figure 7-5.

Most of the non-dormant faults are expected to be caught by the previous mapping check. However, according to Figure 7-5, less than 10% of the faults are detected by previous mapping check. This is due to the methodology of collecting statistics. The assertion check that fires last before the end of simulation is given credit for detecting the fault. A separate statistic for $mcf$ showed that around 75\% of AMT faults were detected by the previous mapping check independently (in the absence of all other fault checks).

Surprisingly, an average of 30\% of the AMT faults is detected by the Ready Bit Status checks. An analysis of the sequence of events that lead to the firing of either of the ready bit assertion checks is insightful. Suppose a fault modifies an entry in the AMT from P1 to P2* (P2 representing the correct version, P2* the faulty version). When P2* is freed from the AMT, it is added to the Free List, which also has P2 as one of its free registers. Suppose I1 and I2 are two instructions whose destination registers are renamed with P2 and P2*. Let I3
be a consumer of I1, with one of its source registers being renamed to P2. Depending on the program order of I2 and I3, either of the ready bit checks fires. It may be recalled that Ready-bit status check I fires when the ready bit of either of the source operands of an instruction is not set when the instruction reaches the Register read stage. In the current scenario, this happens when I3 precedes I2 in program order. The granting of I1 wakes up its consumer I3 in the Issue Queue and sets the ready bit of P2. However, before the consumer I3 is granted from the Issue Queue, the destination of I2 is renamed with P2*, thereby resetting the same ready bit that was set by I1. This causes the firing of Ready bit status check I when the consumer of I1 is granted from the IQ. On the other hand, Ready-bit status check II fires if the ready bit of a register is already set before writing into the register. In the current scenario, this happens if I2 precedes I3 in program order. When I2 is issued, it perceives an already set ready bit of its destination register P2*, thereby firing ready bit check 2. It must be noted that for the above scenarios to occur, both versions of P2, the correct and faulty, need to be located close enough in the Free List to be in flight simultaneously.

The presence of Auxiliary AMT, which is used for reading out the source register mappings, proves to be useful in detecting an average 10% of AMT faults, depicted by source re-renaming in Figure 7-5. The downside of using an Auxiliary AMT is that faults affecting it will raise false alarms of source re-renaming assertion.

Figure 7-5 shows a small percentage of undetected AMT faults for the gap benchmark. Analysis of the scenario leading up to the fault being undetected yielded the following
sequence of events. Suppose that the faulty entry in the AMT is \( L^* \) and the faulty mapping is \( P^* \). The faulty register from the AMT is transferred to the RMT during a load violation recovery. At some future point in time, during the commit of an instruction whose logical destination is \( L^* \), the faulty register \( P^* \) is freed from the AMT and added to the Free List. The timing is not conducive for the fault to be detected by either of the ready bit status checks. Also, the previous mapping check fails because the RMT and the AMT are consistent with each other. The faulty register \( P^* \) is then used as the destination of some instruction, thereby corrupting the contents of \( P \). It should be noted that although this scenario seems highly likely, the number of undetected faults turned out to be only 4 in an experiment of 500 trials, and for only one benchmark. This is because, the presence of two instances of the same physical register will eventually trigger the firing of an assertion, in most of the cases.

### 7.6 Rename Map Table

Prior analysis of Architecture Map Table (AMT) faults helps better understand the statistics for the Rename Map Table (RMT). Figure 7-7 depicts the coverage results for faults in the RMT. The figure shows a high level of fault masking in the RMT, due to the combined effect of register under-utilization and load violations. Faults in the RMT cannot be termed dormant even though the faulty register is unused, since a single load violation is sufficient to mask the fault by overwriting the faulty register. Thus, unlike the AMT, no fault remains in the RMT at the end of the observation period. Other than the masked faults, an average 12% of the faults are benign. The rest of the faults that can cause corruption are all detected, mostly by source re-renaming.
Faults injected into the Shadow Map Tables have a high degree of masking, ranging from a minimum of 94% for gap upto 99% for bzip. This stems from the relatively high branch predication accuracy thereby flushing out most of the faults affecting the SMTs. For the ones that manage to propagate to the RMT, the statistics are similar to those for the RMT.
7.7 Speculative Free List

![Speculative Free List Chart]

**Figure 7-8 Fault injection results for Speculative Free List**

Faults occurring in the Speculative Free List (SFL) manifest themselves as faulty physical destination tags of instructions. Most of these faults are detected by the destination renaming check, as can be observed in Figure 7-8. The Ready-bit status checks also are capable of detecting some faults before the faulty instruction reaches the commit stage. Masking of faults in the SFL occurs when the faulty register tag is overwritten by a new register that is freed from the Architectural Map Table. The degree of masking may also suggest the pattern of physical register utilization by various benchmarks. For example, a higher degree of masking for `bzip` than that for `gzip` may suggest that `bzip` has more number
of physical registers in flight, resulting in a larger number of invalid entries in the Free List at any point in time, compared to gzip.

Another noticeable statistic is the presence of a small percentage of undetected faults for the gzip benchmark. Analysis of the scenario leading to the loss of coverage revealed the following sequence of events. A faulty register in the SFL is used as the destination register of an instruction in the wrong path, thereby corrupting the value of the register. The correct version of the register is used as the source by a branch instruction to calculate its target PC. The branch instruction thus gets a nonsensical target address thereby steering the processor to fetch NOP instructions. Even though the faulty register is added back to the Free List during branch recovery, it is never used again since the renaming process effectively stops due to the fetching of NOPs. However, in a real processor, the renaming process never stops as the processor fetches valid (though, incorrect) instructions. Thus, the presence of two versions of the same physical register tag will trigger at least one of the fault assertions.

**7.8 Issue Queue Payload**

The targeted field in the Issue Queue payload is the physical destination register of the instruction. The statistics shown in Figure 7-9 are very similar to those of the physical destination field of Issue-Register read pipeline latch. The ready-bit status check II is capable of detecting most of the faults, while the rest are detected by destination re-renaming. It can be observed that gap and vortex have a small percentage of benign faults, indicating that an incorrect destination register did not result in architectural corruption.
7.9 Issue Queue Source Ready Bits

Figure 7-10 shows the coverage of faults affecting ready bits of physical source registers of instructions waiting to be issued in the Issue stage. As can be deduced from Figure 7-10, an average of 25% of the faults on ready bits are masked and 25% of them are benign. A source ready bit fault turns out to be benign in cases where the fault incorrectly sets the ready bit of a source operand, yet the instruction issue is delayed long enough till the source register is actually ready when it issues. Ready bit status check I detects faults that incorrectly set the ready bit of the source operand causing an immature issue of the instruction. The watchdog timer detects faults that reset the ready bit of the source operand, resulting in a state of deadlock.
Figure 7-10 Fault injection results for the source ready bits in Issue Queue

7.10 Issue Queue CAM

Figure 7-11 Fault injection results for the physical source register CAM in Issue stage
The Issue Queue CAM houses the physical source registers of instructions in the Issue Queue. As depicted in Figure 7-11, an average 35% of the faults are caught by source re-renaming. About 30% of the faults cause instructions to wait on phantom source registers, thereby forcing the processor to enter a state of deadlock. Such faults are detected by the watchdog timer.

### 7.11 Active List

The Active List stores the physical and logical destination registers of instructions until they retire. Figure 7-12 shows the coverage of faults affecting these fields in the Active List. In addition to these fields, the Active List also stores the logical source registers and previous RMT mapping for every instruction as part of the fault check regimen, as discussed Chapter 5. Faults affecting these additional fields in the Active List and those affecting the Auxiliary Active List do not affect program outcome, and hence are benign faults. However, they do cause false assertion failures to be raised.

As shown in Figure 7-12, all faults affecting the physical destination tags of instructions are detected by destination re-renaming during instruction retirement. Those affecting the logical destination registers are detected by the previous RMT mapping check. As in other structures, an average 20% of the faults are masked due to flushing of the Active List, caused due to either branch mispredictions or load violations.
In conclusion, this chapter presented the results obtained from fault injection on various structures in a superscalar processor holding logical and physical register tags. The unified regimen successfully detected more than 99% of the non-masked faults on targeted structures in the observed simulation period. A small percentage of undetected faults affecting the Speculative Free List, the Architectural Map Table and the Execute-Writeback pipeline register were analyzed and were found to occur due to a very specific sequence of events occurring very rarely during normal execution.
Chapter 8 : RTL Synthesis Results

This chapter discusses the overhead due to the new fault check regimen on the baseline microprocessor, obtained by synthesizing the RTL model. It also discusses the processor silicon area coverage of the new fault check regimen.

8.1 Area Overhead

This section discusses the area overhead of each of the microarchitecture-level fault checks comprising the unified fault check regimen, discussed in Chapter 5.

(a) Source re-renaming: The structural additions necessary for the implementation of source re-renaming are:

- Additional fields in the Rename-Dispatch pipeline register for logical source tags [10 X 4 bits]
- Additional fields in the Execute-Writeback pipeline register for physical source tags [14 X 4 bits]
- Additional fields in the Auxiliary Active List for physical source tags [14 X 128 bits]
- Additional fields in the Active List for logical source tags [10 X 128 bits]
- Auxiliary Architectural Map Table (Ax AMT) [ SRAM: 8R4W; 7 X 32 bits]
- Logic for checking dependencies among sources and destinations of instructions retiring in the same clock cycle
- Comparison logic, consisting of 8 7-bit comparators

(b) Destination re-renaming: The additions for performing destination re-renaming during retirement are:
- Additional field in the Auxiliary Active List for physical destination tags [7 X 128 bits] (Not necessary if already part of the existing implementation)
- Architectural Free List [SRAM: 4R4W; 7 X 96 bits]
- Comparison logic, consisting of 4 7-bit comparators

(c) Ready-bit status checks: The ready bit status checks do not require any major hardware addition, except for the small combinational logic associated with the physical register ready bit array.

(d) Previous mapping check: The implementation of previous mapping check imposes the following hardware additions:
- 4 additional read ports on the Rename Map Table, to read out the mappings before they are updated
- Additional fields in the Rename-Dispatch pipeline register for previous mappings [7 X 4]
- Additional fields in the Active List for previous mappings [7 X 4]
- Comparison logic in the retirement module, consisting of 4 7-bit comparators

(e) Watchdog timer: The watchdog timer comprises of a 10-bit down counter and associated combinational logic, which do not contribute to the area significantly.
The RTL model of the superscalar processor incorporating the new fault check regimen was synthesized using the Nangate 45nm Open Cell Library. The area overhead over the baseline core due the regimen was found to be approximately 5.8% (excluding the caches). The areas of RAM based structures were obtained using [32].

8.2 Die Area Coverage of Fault Check Regimen

Breakup of the die area occupied by various pipeline stages of the superscalar processor is shown in Figure 8-1.

The new fault check regimen covers approximately 12% of the die area of the superscalar processor. This includes the Rename stage, the Issue stage, the Commit stage and specific fields in pipeline registers. These modules form the heart of the complex dynamic control logic of the microprocessor, and hence are worth the small overhead necessary in order to protect them. A large percentage of the die area deals with static information, which can be effectively and efficiently protected using techniques such as Inherent Time Redundancy (ITR) [33]. Hence, a comprehensive regimen, comprising of the microarchitecture-level fault checks introduced in this thesis and the ITR technique, will be capable of protecting most of the die area of a superscalar processor and detect faults in both, static as well as dynamic information within the processor. Incorporation of ITR into the fault check regimen has been left for future work.
Figure 8-1 Area breakup of pipeline stages in a superscalar processor
Chapter 9: Conclusion and Future Work

9.1 Conclusion

Considering that physical register tags form an important component of the dynamic control information whose correctness is paramount to the correct execution of a program, this thesis, firstly, analyzed the vulnerability of physical register tags in the various pipeline stages of a superscalar processor. Secondly, this thesis extended a previously proposed methodology to detect faults in a superscalar processor and proposed a new regimen comprising of microarchitecture-level fault checks, to detect faults in register tags throughout a superscalar processor. The fault check regimen was designed to be effective, by providing a comprehensive end-to-end coverage of faults affecting register tags, and efficient, by minimizing the overhead to the baseline core. The regimen was designed and analyzed at the Register Transfer Level to determine its fault coverage and its overhead to the baseline core. Extensive fault injection experiments were performed on the RTL model of a superscalar processor incorporating the fault check regimen, and determined its fault coverage to be more than 99% for the targeted structures. In addition, the area overhead of the fault check regimen on the baseline core was determined to be less than 6%. In conclusion, the newly proposed fault check regimen is capable of detecting faults in physical register tags throughout the entire pipeline, thereby covering critical units, including the register rename logic, the issue logic and the retirement logic, and critical structures such as the Rename Map Table, the Architectural Map Table, the Issue Queue and the Free List.
9.2 Future Work

Firstly, the proposed regimen could be extended to protect static information in all pipeline registers, and thereby protect other pipeline stages in a superscalar processor. Inherent Time Redundancy (ITR) could be effective employed for this purpose, as proposed by [33]. Secondly, the mechanism required to recover from faults that are detected needs to be studied and implemented, and thereby estimate the performance implications arising due to the firing of false assertions by the fault check regimen.
REFERENCES


Champaign, 2006.


