The Pennsylvania State University
The Graduate School
College of Engineering

ADDRESSING RELIABILITY ISSUES IN PERFORMANCE-CRITICAL PROCESSOR STRUCTURES

A Dissertation in
Computer Science and Engineering
by
Niranjan Soundararajan

© 2010 Niranjan Soundararajan

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

August 2010

The dissertation of Niranjan Soundararajan was reviewed and approved* by the following:

Anand Sivasubramaniam
Professor of Computer Science and Engineering
Dissertation Co-Adviser, Co-Chair of Committee

Vijaykrishnan Narayanan
Professor of Computer Science and Engineering
Dissertation Co-Adviser, Co-Chair of Committee

Mary Jane Irwin
Evan Pugh Professor of Computer Science and Engineering

Yuan Xie
Associate Professor of Computer Science and Engineering

Suman Datta
Monkowski Associate Professor of Electrical Engineering

Raj Acharya
Professor of Computer Science and Engineering
Head of the Department of Computer Science and Engineering

*Signatures are on file in the Graduate School.

Abstract

Diminishing transistor sizes combined with power and performance constraints have decreased the inherent robustness in devices. With each new generation, designers find it difficult to provide deterministic operating characteristics or guarantee a certain lifetime. This has led to the need for architectural designs that not only need to be performance-efficient but also provide reliable operation as well. This dissertation is a step towards developing architectural designs that address reliability issues.

As a case in point, we look at two performance-critical processor structures - the issue queue and reorder buffer. Given that devices can fail at any point in their lifetime, our solutions address failures occurring at these different periods. These include manufacturing process variations that affect the operating speeds of the device, soft errors that flip the stored data value and wearout failures that severely limit the useful operating period of a device.

Issue queues in high performance microprocessors involve multiple activities, moving instructions in and out of the queue, occurring within a single cycle. Process variation in the issue queues limit the synchronization that exists amongst these different activities in turn limiting their operating frequency. Our solution allows the faster and slower cells in the issue queue to co-exist limiting the stall cycles even at higher frequencies, thereby allowing performance scaling to continue. Soft errors occur when random particles strikes cause a transistor state to flip. Complete protection against soft errors is a overkill for certain market segments as this protection comes at the cost of performance. We provide microarchitectural solutions that provide guarantees on limiting the soft error vulnerability of a platform based on limits required by the market. Our solution enables flexible spanning of the performance-reliability space depending on requirements. Further, we explore the impact of soft error vulnerability in multicores. We do a detailed analysis and identify factors that determine the overall vulnerability when the number of

cores gets scaled and use this analysis to provide a runtime solution to minimize the soft error vulnerability. Also prior works have ignored the soft error vulnerability when scaling the voltage and frequency of a system. Our analysis clearly shows the need for designers to be aware of the reliability impact when adopting different voltage frequency algorithms. Finally, we address wearout issues in the issue queue which lead to entries getting shut down early in a microprocessor's lifetime, in turn affecting the overall performance. Our solutions to limit the variation in wearout in the issue queue entries decrease the degradation significantly and allow its entries to age more uniformly. This in turn helps them meet lifetime requirements.

Together, this dissertation provides a comprehensive framework minimizing the impact of failures at different points in a device's lifetime. This work is a step towards developing fault-tolerant architectural designs that are performance- and cost-competitive across all market segments.

Table of Contents

List of Figures **ix**

List of Tables **xiii**

Acknowledgments **xiv**

Chapter 1
Introduction **1**
1.1 Hardware Failure Classification 2
1.1.1 Process Variations 3
1.1.2 Soft Errors 4
1.1.3 Wearout 4
1.2 Components of Interest in an Out of Order Pipeline 5
1.3 Organization of the dissertation 6

Chapter 2
Analysis and solutions to Issue Queue Process Variation **7**
2.1 Introduction 7
2.2 Background 9
2.3 Simulation Setup 11
2.4 Variation analysis on the key issue queue components 12
2.5 The effects of PV on issue queue operation 13
2.6 Battling PV artifacts within the issue queue 14
2.6.1 PV-aware instruction steering 15
2.6.1.1 SpeedSteer: A Speed-aware steering mechanism . . 16
2.6.1.2 OptiSteer: An optimized table-based steering mechanism 16

2.6.2 Intra-Entry variations . . . 18
2.6.2.1 Operand Switching . . . 18
2.6.2.1.1 Implementation . . . 20
2.6.2.2 Port Switching . . . 20
2.6.2.2.1 Implementation . . . 21
2.6.3 Existing mechanisms for handling variations . . . 22
2.7 Epilogue to PV-aware issue queue design . . . 24
2.7.1 Microarchitectural support for pipeline stalling . . . 24
2.7.2 Variation testing methodology . . . 25
2.8 Related Work . . . 26
2.9 Summary . . . 28

Chapter 3
Mechanisms for Bounding Vulnerabilities of Processor Structures 29
3.1 Introduction . . . 29
3.2 Related Work . . . 30
3.3 Background and Motivation . . . 33
3.3.1 Measuring System Vulnerability . . . 33
3.3.2 Architectural Vulnerability Factors . . . 33
3.4 Meeting Reliability Budgets . . . 34
3.4.1 Bounding Architectural Vulnerability Factors . . . 34
3.5 Dynamic Vulnerability Monitoring . . . 35
3.5.1 Baseline Microarchitecture . . . 36
3.5.2 Identifying ACE bits . . . 36
3.5.3 Online AVF monitoring infrastructure for the ROB . . . 36
3.6 Vulnerability Control via Throttling (VCT) . . . 38
3.7 Vulnerability Control via Selective Redundancy (VCSR) . . . 40
3.7.1 Overview . . . 40
3.7.2 Achieving Redundant Execution . . . 41
3.7.3 Achieving Selective Redundancy . . . 42
3.7.3.1 Maintaining a Consistent Architected State . . . 43
3.7.4 Selecting Instructions for Redundant Execution . . . 44
3.7.4.1 Impact of Mis-Speculation . . . 45
3.7.5 Hybrid Vulnerability Control (VCH) . . . 46
3.8 Optimizations . . . 46
3.8.1 Out of Order Commit . . . 47
3.8.2 OoOC and AVF reduction . . . 48
3.8.3 Ensuring Correct ARF updates with OoOC . . . 48
3.8.4 OoOC without Collapsing (OoOC-NC)c . . . 48
3.9 Results . . . 49

3.9.1 VCT Results 50
3.9.2 VCSR Results 53
3.9.3 VCH Results 54
3.9.4 Non-Collapsing OoO Commit 55
3.10 Summary 57

Chapter 4
Multicore Soft Error Vulnerability Characterization **58**
4.1 Introduction 58
4.2 Experimental Methodology 61
4.2.1 Simulation Infrastructure 61
4.2.2 System Configuration 62
4.2.3 Multicore AVF Analysis Framework 62
4.3 Multicore ROB AVF Analysis 64
4.3.1 AVF Breakdown for the 8-core CMP 65
4.3.2 Factors affecting AVF variation 66
4.3.2.1 Instruction Characteristics 67
4.3.2.2 Impact of coherency misses 68
4.3.2.3 Impact of spinlocks 69
4.4 Impact of Multicore parameters 71
4.4.1 System AVF and System Vulnerability 72
4.4.2 Performance Impact 74
4.4.3 Impact on System AVF 74
4.4.4 Impact on System Vulnerability 75
4.5 Core-Allocation schemes to optimize Performance Reliability tradeoffs 76
4.5.1 Static Spinlock 78
4.5.2 Static MITF 78
4.5.3 Adaptive MITF 79
4.6 Related Work 80
4.7 Summary 82

Chapter 5
Impact of Dynamic Voltage Frequency Scaling on the Architectural Vulnerability of GALS Architectures **83**
5.1 Introduction 83
5.2 Background 85
5.3 DVFS Algorithms 85
5.4 Simulation Setup 88
5.5 Experimental Results 89
5.5.1 AVF Variation across DVFS algorithms 90

5.5.2 Performance-Power tradeoffs . . . 91
5.5.3 Performance-Vulnerability tradeoffs . . . 91
5.5.4 Power-Vulnerability optimization . . . 92
5.5.5 Vulnerability Efficiency characterization . . . 92
5.5.6 Impact of voltage scaling on overall FIT rate . . . 93
5.5.7 Sensitivity Analysis . . . 95
5.6 Summary . . . 96

Chapter 6
Mechanisms to address Non-uniform aging **97**
6.1 Introduction . . . 97
6.2 Background . . . 100
6.2.1 Impact of NBTI and HCE . . . 100
6.2.2 Baseline Microarchitecture . . . 101
6.3 Motivation . . . 102
6.4 Simulation setup . . . 103
6.5 Mitigating switching activity variation in collapsible issue queues . . . 105
6.5.0.1 Design for restricted collapsing in Issue Queues . . . 106
6.6 Related Work . . . 109
6.7 Summary . . . 111

Chapter 7
Conclusion and Future Work **112**
7.1 Conclusion . . . 112
7.2 Future Work . . . 114

Bibliography **115**

List of Figures

1.1 Hardware failure rate over time, courtesy [8]. 3
1.2 Conventional out-of-order pipeline. The front-end is in-order while the back-end of the pipeline is out-of-order. 5

2.1 (A) Out-of-Order Pipeline highlighting the paths to the Issue Queue. The Figure also shows an Issue Queue entry and all associated activities. (B) Time-line of issue queue activities at cycle-level granularity. Here 't' refers to a cycle. The cycles are not drawn to scale. 9
2.2 Delay variation in issue queue. (C) shows that 60% of entries operate in 1 cyle while 40% of entries require 2 cycles. 13
2.3 Performance Impact of slowing down different activities that occur in the Issue Queue. Legend shows the average IPC across all benchmarks and the high IPC ones. 15
2.4 Microarchitectural changes required to implement OptiSteer. 15
2.5 Performance obtained by employing the instruction steering schemes. Legend shows the average IPC across all and high IPC benchmarks clearly showing the requirement for the steering schemes to reduce performance loss due to variations. 17
2.6 Percentage of single-operand instructions and ready operands read out of register file. Legend shows average value across all benchmarks. 19
2.7 Bus utilization for dispatch, issue and forwarding. Average value shows that the effective utilization is low. 20
2.8 Port-switching logic implementation. (A) Overall connections between buses available and port speed lines. (B) Logic inside each of boxes assigning either one of the buses based on availability and port speeds. 21
2.9 Performance impact of combining steering with operand and port-switching. Legend clearly shows that across all benchmarks and high IPC ones, the hybrid scheme does well. 23

2.10 Performance Degradation due to shutting down variation-affected entries. Average values across all benchmarks and high IPC ones show that there is significant performance loss with shutting down entries. 23
2.11 Performance obtained by operating the Issue Queue in the two modes in a MCD microarchitecture. The average values show that this might not be an effective solution given the performance loss incurred. 24
2.12 Implementation of the stall logic in the different issue queue operations. 25
2.13 Time-line of our mechanisms and stall handling at cycle-level granularity. Here 't' refers to a cycle. The cycles are not drawn to scale. 26

3.1 Dynamic AVF Variation in *177.mesa* at Cycle (Left) and 100 Cycles (Right) Granularity 31
3.2 Online AVF monitoring infrastructure for the ROB. 37
3.3 VCSR overview. 42
3.4 IPC with VCT for various AVF bounds. Also provided for each benchmark are IPCs for single-thread execution (rightmost bar) and SRT (leftmost bar). 50
3.5 CDF of the cycle granularity AVF over the single-thread execution. 51
3.6 (A) AVF Estimation of the ROB by VCT averaged over all the cycles highlighting the conservatism in throttling. (B) Percentage of instructions redundantly executed with VCSR for different AVF bounds. 52
3.7 IPC with VCSR for various AVF bounds. Also provided for each benchmark are IPCs for single-thread execution (rightmost bar) and SRT (leftmost bar). 53
3.8 Summary of Results. Geometric Mean of IPC across Applications Normalized w.r.t. Single Thread IPC for different AVF bounds. Also shown is SRT performance on the y-axis (i.e. AVF=0). 55
3.9 IPC with VCSR using Non-collapsing OoO Commit 56

4.1 Pipeline design within each core. Figure highlights pipeline events specific to the Reorder Buffer (ROB) besides showing sub-components within a ROB entry. 62
4.2 AVF variations across different cores of 8-core CMP. The height of the bar corresponds to the overall AVF of a core. 65

4.3 MITF variations across different cores of 8-core CMP. The significant variation in MITF across cores imply a need for protection techniques to be aware of these variations to provide better performance-reliability tradeoffs. 66
4.4 ACE/unACE instruction classification. The height of each bar corresponds to the fraction of the total instructions that retire from a core. . 67
4.5 Fraction of the overall AVF due to instructions (user and kernel) in shadow of L1 coherency miss. The values are averaged across the 8 cores. 69
4.6 Fraction of kernel AVF due to spinlocks. The solid portion in each bar gives the unACE portion of the "test and test-and-set" code. . . 70
4.7 System AVF and System Vulnerability variations across three different multicore configurations when running fma3d. The figure shows that while the System AVF is highest for a configuration(8T_8C) it does not manifest in a higher System Vulnerability, which is highest for 2T_2C. . 72
4.8 Execution time variation across the different multicore configurations. . 73
4.9 System AVF across the different multicore configurations. Solid portion gives the kernel code contribution while striped portion is the user code contribution. 75
4.10 Vulnerability across the different multicore configurations. Solid portion gives the kernel code contribution while striped portion is the user code contribution. 75
4.11 Figure shows the runtime variation in throughput (aggregate IPC across cores) and System Vulnerability when executing LU in 2 configurations. It highlights phases that can be exploited for decreasing System Vulnerability without losing performance. In (A)Throughput Difference > System Vulnerability Difference. Hence 8T_8C is more optimal in this phase. In (B)System Vulnerability Difference > Throughput Difference. Hence 8T_4C is preferred in this phase. 77
4.12 Execution time comparison across configurations revealing the low overhead of the core allocation schemes. 78
4.13 System Vulnerability comparison across configurations. Compared to 8T_8C the core allocation schemes reduce System Vulnerability by greater than 35% (MITF schemes). 79
4.14 System AVF comparison across configurations. Given that AVF cannot be significantly increased, these graphs show the core allocation schemes to have only a moderate impact on the System AVF compared to the 8T_8C configuration. 80

5.1 GALS pipeline showing the different asynchronous domains 85

5.2 Impact of different DVFS algorithms on issue queue's AVF. 89
5.3 Performance Vulnerability tradeoffs of the DVFS algorithms. 90
5.4 Performance per Watt of the DVFS algorithms. 91
5.5 Power Vulnerability tradeoffs of the different DVFS algorithms. . . 92
5.6 Vulnerability Efficiency of the different DVFS algorithms. 93
5.7 Average AVF variation for GALS pipeline with 4 and 5 asynchronous domains 94
5.8 Performance, Power and Reliability tradeoffs provided by different DVFS algorithms 95
5.9 (A) Fraction of time each DVFS algorithm spends in a VF level. (B) Linear raw error rate voltage relationship. (C) Exponential raw error rate voltage relationship. 95

6.1 (A) Base pipeline we use in this study. (B) Performance loss due to shutting down entries shown for a SPEC2K benchmark(*art*) 101
6.2 Switching activity variation across ISQ. 102
6.3 (A) Cell design in latch-based collapsible queues. (B) Distribution showing collapsing queue entry degradation as a function of data, switching activity and writes. 105
6.4 (A) Conventional collapsing logic design. (B) Modifications to support restricted collapsing. 106
6.5 (A)Switching activity variation across ISQ entries. (B)Performance impact of Restricted Collapsing. 107
6.6 ISQ read delay degradation. 108

List of Tables

2.1 Simulation parameters. Latencies of ALUs/caches are given in parentheses. All ALU operations are pipelined except division and square-root. 11

3.1 Simulation parameters. Latencies of ALUs/caches are given in parenthesis. All ALU operations are pipelined except division and square-root. 49

4.1 (A) Per-core and multicore platform parameters. (B) Benchmarks used in our analysis 61
4.2 Different configurations used in our evaluation. 71
4.3 Different parameters associated with core-allocation schemes 76

5.1 Simulation parameters. 88

6.1 Simulation parameters. Latencies of ALUs/caches are given in parentheses. All ALU operations are pipelined except division and square-root 103

Acknowledgments

It has been a long journey and I have got several people to thank for helping me out over these years. First of all, I would like to thank God for being my strength during my most difficult times.

I would like to thank my advisers, Dr.Anand and Dr.Vijay, for helping me at different times in my PhD. Both are super-enthusiastic about discussing research and, whenever I am stuck, are great in directing me to take the project forward. Both have guided me on how to conduct research and more importantly present the work in a lucid manner. I am very thankful to both for having supported me financially over these years.

The Computer Science and Engineering department at PSU offers a great program with good courses for someone working in computer architecture. I am happy to have interacted with Professors Mary Jane Irwin and Yuan Xie at various points over my five years. Their inputs has been valuable in shaping me. I would like to thank all my committee members: Professors Anand Sivasubramaniam, Vijay Narayanan, Mary Jane Irwin, Yuan Xie and Suman Datta for their help and feedback in writing my dissertation and allowing me to flexibly schedule my comprehensive exam and final oral defense.

I have been fortunate to have interacted with folks from two labs - Microsystems Design Lab (MDL) and Computer Systems Lab (CSL). Over the years the talks that I have attended and the discussions I have had definitely helped in providing a perspective on where my works fall and how best to present them. I would like to thank Angshuman Parashar with whom I worked on my first project. As a senior in the lab, his knowledge in architecture was very helpful in getting me started. Others in the labs, including Shiva Chaitanya, Sriram Govindan, Ramakrishnan Krishnan, Suresh Srinivasan, Balaji Vaidyanathan and Aditya Yanamandra have all helped me at various times. There are definitely more people with whom I have interacted with and I am thankful to all of them as well.

I would like to definitely thank my under-graduate adviser, Prof. Venkateswaran, for giving me an opportunity at WARFT. The opportunity to interact with like-

minded peers was greatly helpful in motivating me to do a Ph.D.

Finally, I would like to thank my parents, brother, grand-parents and family for standing by my side and providing the emotional support. Their constant encouragement and motivation was greatly helpful through my five years.

Dedication

I would like to dedicate this work to my family who have stood by me through all my life and constantly been a motivation. I have learned great things from them and hopefully will get a chance to give back as well.

Chapter 1

Introduction

Performance, power and reliability are the three dimensions of transistor design. Depending on the market segment and design constraints, each of these dimensions assume primary importance. Over many technology generations, till now, transistors have been robust and reliable operation has been obtained almost free of cost. Designers, therefore, have looked to address performance, power and on-chip thermal issues. The emergence of metrics like performance per watt (PpW) and thermal design point (TDP) indicate the importance given by architects to these metrics.

As we enter into the sub-25 nanometer(nm) regime[48], a multitude of physical phenomena and design requirements are starting to affect transistor reliability [72, 53]. No longer can designers assume reliable operation over sustained periods of time. This has given rise to the need for fault-tolerant and reliable hardware. The factors that have led to this situation can broadly be given as

- **Decreasing Transistor size:** The continuous drive towards meeting Moore's law has shrunk the world of microprocessors considerably. We are getting ready to enter the sub-25 nm regime [48]. As such, device manufacturing will not only be limited by the shrinking device sizes but also hit a roadblock in terms of producing these at manageable costs [96]. We can no longer expect devices to behave in a deterministic manner in accordance with their design.

- **Explosion in on-chip transistor count:** Core scaling has replaced frequency scaling [54, 58, 50, 35]. As such, the growing number of on-chip cores

has led to a significant increase in on-chip transistors. We have reached the multi-billion transistor era [127]. This places significant constraint on the overall power envelope of a chip. To tackle this requirement and the decreasing transistor sizes, designers have looked at supply voltage (V_{DD}) scaling. But decreasing V_{DD}, has led to a loss in the inherent device robustness and they are more prone to the noise that exist within a chip [29].

- **Thermal Hotspots:** The emergence of multicores has led to a increase in thermal hotspots on a chip. Temperatures reaching 100 °C are possible at several locations within a chip [85, 77]. Sustained operation at these temperatures dramatically changes the transistor characteristics leading to failures in meeting lifetime requirements.

- **Growing market requirements:** While the emergence of newer market segments has no direct impact on the chip reliability, the decreasing time to market constraints has led to using the same underlying architecture across several designs [84]. The differing power and performance requirements, across the different market segments, severely constrain the underlying hardware and affect their capability to provide reliable operation.

1.1 Hardware Failure Classification

This section provides a brief description on the different failure mechanisms that cause reliability concerns in current and future generation processors. Hardware faults, typically, get classified as hard (permanent), intermittent or transient [31]. Permanent (or hard) faults reflect changes that are irreversible in the underlying circuitry. Transient faults are induced by cosmic rays or electromagnetic interference. Intermittent faults occur due to unstable hardware.

Failures due to these faults occur at various points in a transistor lifetime. Figure 1.1 gives a typical bath-tub curve associated with transistor failure rates. The failures can be classified into three categories. Early infant mortality occurs due to manufacturing defects and the failure rate due to these phenomena decrease with use over time. On the other hand, wearout failures, due to transistor aging, increase over lifetime. Besides failures due to these phenomena, transistors can

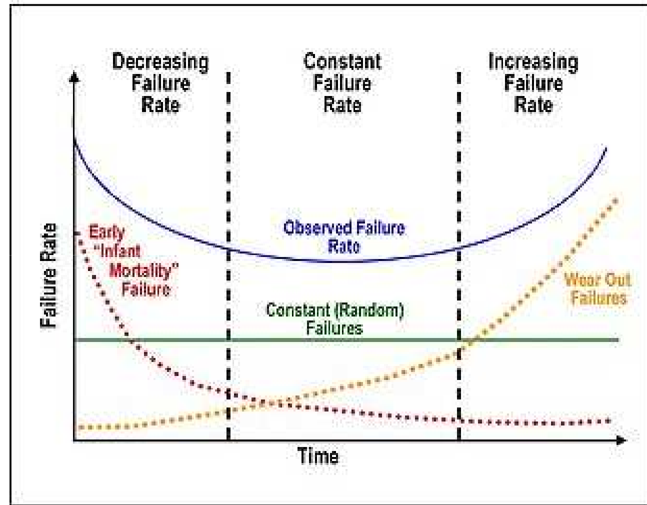


Figure 1.1. Hardware failure rate over time, courtesy [8].

also fail due to random extraneous events that affect their state and corrupt the stored values. Given that they are equi-probable to occur at any time during a transistor's lifetime, the failure rate is assumed to be constant over time. With the increasing reliability concern, failure rates in all three intervals have risen. Amongst the different reliability phenomena that affect transistor operation and inhibit them form operating reliably, manufacturing process variations, random soft errors and wearout due to Negative Bias Temperature Instability (NBTI) and Hot Carrier Effects (HCE) have gained significant attention from architects [6, 11]. This dissertation would look at addressing reliability issues arising due to all three phenomena.

1.1.1 Process Variations

Process variations (PV) are a major factor in early lifetime failures of transistors. Semiconductor manufacturing variations lead to deviation in device characteristics between identically designed circuits and hence prevent designers from making assumptions on their behavior. While previously, burn-in tests were used to separate out circuits that did not match specific operating requirements, technology scaling has accentuated process variations which in turn have reduced the effectiveness of these tests. No longer is frequency binning [113] of dies sufficient to separate chips meeting different operating speeds. Process variations affect several parameters of

transistor design including gate length, inter layer dielectric thickness and gate oxide thickness [19]. These variations in turn manifest as variations in the on-current (I_{ON}) and threshold voltage (V_{TH}) which directly affect the switching frequency of the devices and their leakage characteristics.

1.1.2 Soft Errors

Radiation induced soft errors are transient faults that cause a transistor state to flip when high energy neutron particles from cosmic rays or alpha particles from the transistor substrate strike them. The particle strike produces electron-hole pairs in the substrate resulting in a current spike in the circuit resulting a transient fault. Together with crosstalk between wires, radiation-induced soft errors form the primary source of transient failures. As all components within the chip are equally prone to radiation-induced soft errors, replacing the circuitry does not improve robustness.

1.1.3 Wearout

Continuous operation of a chip alters the underlying transistor timing and functionality over time. With the rise in power and thermal hotspots on a chip, different parts of the chip operate under differing conditions which affects their overall lifetime. Combined with transistor scaling, the impact of transistor lifetime has been dramatic leading to circuits that fail much earlier than the guaranteed lifetime. Time Dependent Di-electric Breakdown (TDDB), Negative Bias Temperature Instability (NBTI) and Hot Carrier Effects (HCE), also known as Hot Carrier Injections (HCI), and Electro-Migration (EM) are four dominant wearout phenomena.

In this dissertation, we primarily concentrate on NBTI and HCE, whose effects are aggravating in the 45nm and smaller device generations [72, 53]. NBTI is predominant in PMOS transistors while HCE is predominant in NMOS transistors. These two mechanisms cause a change in the threshold voltage (V_{TH}) of the device thereby affecting their operating speed. Huge changes in the V_{TH} of the device increase the delay of the circuit leading to a timing failure.

1.2 Components of Interest in an Out of Order Pipeline

Failures occur at different locations on a chip. While the overall goal of resilient chip design is to address all the issues, this dissertation addresses reliability issues in two structures that are very critical for high performance within a microprocessor's pipeline, the Issue Queue and Reorder Buffer.

Figure 1.2 shows a typical out-of-order pipeline. Instructions are fetched and decoded in-order in the front-end of the pipeline. Register renaming occurs in the Register Alias Table (RAT), which maintain tags corresponding to the destination architected registers. The Reorder Buffer (ROB) and Physical Register File (PRF) are coupled together and hence the ROB entries correspond to PRF ids. The Architectural Register File (ARF) exists as a separate structure in which instructions retiring from the ROB write their results.

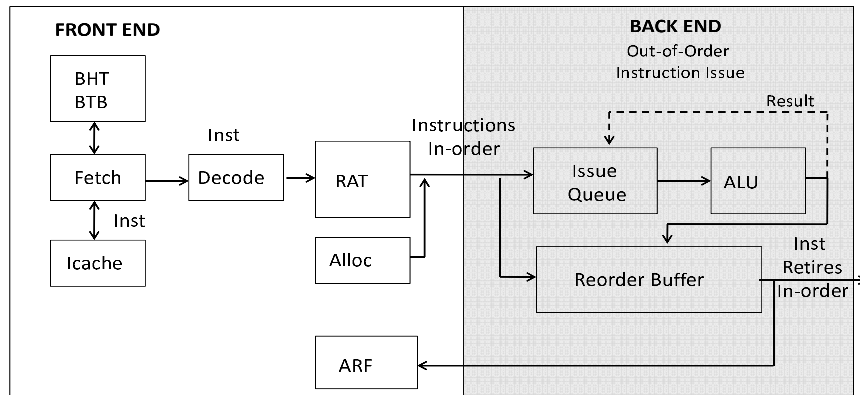


Figure 1.2. Conventional out-of-order pipeline. The front-end is in-order while the back-end of the pipeline is out-of-order.

The front-end is in-order, with respect to handling instructions in program order, while the back-end is where the instruction level parallelism (ILP) is extracted. The back-end allows instructions to execute out-of-order (OOO) enabling tremendous increase in performance. Two specific structures that enable this out-of-orderness are the issue queue and reorder buffer.

The issue queue (ISQ) receives instructions in-order after they get decoded. It allows instructions to proceed to their execution units depending on data dependency, independent of the program order. This is enabled by holding tags corre-

sponding to the data operands that the instructions depend on. Once available, the tags are flagged and the instructions can issue.

The ROB is required to maintain the correct system state. Retiring instructions that complete OOO can lead to inconsistent system state. Doing so would lead to exceptions not getting handled in the correct order. Further, mis-speculated state that needs to be flushed out the system require the ROB to maintain instructions in-order to enable correct execution.

Designers optimize these two structures to extract the maximum instruction level parallelism (ILP) in the smallest cycle time. As such, these structures are stretched to the edge of their reliable operation. This dissertation looks at providing architectural solutions that enhance their reliable operation. Any protection mechanism comes at the cost of performance overhead. Therefore, these solutions have been proposed with an eye towards minimizing the performance impact as well.

1.3 Organization of the dissertation

We first look at solutions to address the impact of process variations in the issue queue in chapter 2. Chapter 3 proposes different mechanisms to bound the soft error vulnerability and providing guarantees on market specified reliability requirements. Chapter 4 provides a detailed study on the impact of soft errors in multicores incorporating both application and kernel codes. Using this analysis, we provide solutions to minimize the soft error vulnerability of a multicore system. Dynamic Voltage Frequency Scaling (DVFS) algorithms are typically evaluated using performance per watt. Chapter 5 looks at their impact on soft error vulnerability. Chapter 6 addresses wearout issues in the issue queue. Finally, chapter 7 concludes the dissertation summarizing the work and discusses future work.

Chapter 2

Analysis and solutions to Issue Queue Process Variation

2.1 Introduction

The need for more computing power and smaller platforms has led the IC technology towards minuscule feature sizes. The latest chips are starting to surpass the billion transistor mark [16]. This push towards ever-increasing transistor counts has given rise to new challenges and impediments. In particular, Process Variation (PV) [16, 19] has become a major design consideration. PV arises from manufacturing imperfections resulting from sub-wavelength lithography, Random Dopant Fluctuations (RDF), dose, focus, and overlay variations [116]. The aggravation of such manufacturing uncertainties invariably leads to marked deviations in effective gate length, oxide thickness, and transistor threshold voltages [19]. Non-nominal device characteristics may lead to substantial variations in power consumption and timing violations. PV can be a systematic or a random phenomenon. While systematic variations exhibit strong spatial correlations, such that structures close to each other are affected similarly, random variations can occur anywhere.

PV creates a widening gap between designed and manufactured circuit characteristics. This disconnect adversely affects the chip yield leading to circuit design becoming more probabilistic in future. Several researchers have already looked into the severity of the problem and proposed architectural-level and circuit-level

solutions to mitigate its effect [70, 116].

The chapter focuses on one of the key components of any modern microprocessor, namely the Issue Queue. Owing to its complex microarchitecture and its importance in determining the overall performance of the system, issue queues necessitate a careful evaluation and assessment of their vulnerabilities to PV. It will be demonstrated here that the nominal-value deviations imparted by PV could potentially lead to severe degradation in system performance. Designing for worst-case scenarios is certainly not feasible due to the issue queue's significant impact on performance.

Developing effective solutions to guard against PV artifacts in issue queues is non-trivial. For instance, since the issue queue power density is high, conventional techniques such as Forward Body Biasing (FBB) [5] are not suitable for mitigating PV effects. Unlike other structures studied for variations [67], the issue queue involves multiple activities which include instruction dispatch, issue and forwarding that together make it challenging to synchronize these activities under variation. Summarizing the major contributions:

- An in-depth analysis of the issue queue operation.As such, various issue queue activities that are susceptible to variations are identified and the impact of slowing them down is analyzed. We show that PV-unaware issue queue operation can lead to 20.5% performance degradation compared to a conventional PV-unaffected system.

- Mechanisms to counter variations have been provided. Since variations affect the entries in a non-deterministic manner, our techniques enable fast and slow entries within the issue queue to co-exist and thereby allow instruction dispatch, issue and forwarding operations to synchronize with each other. These mechanisms bring down the performance degradation to 5%.

- Sub-components within the issue queue entry vary in their operating speed as well. Switching source operands of instructions based on their availability is investigated. Further port-switching, whenever possible, is also shown to be effective. This sub-component analysis further reduces the performance degradation to only 1.3%, highlighting the efficacy of our mechanisms to counter variations.

Section 2.2 provides a background on issue queue design and associated pipeline activities. Section 2.3 gives the simulation platform while section 2.4 discusses the impact of variations with respect to the CAM and SRAM cells within the issue queue. Section 2.5 talks about the performance impact of slowing down different issue queue activities. Section 2.6 provides solutions to handle slow entries in the issue queue and reduce their performance impact. Section 2.7 talks about the testing methodology and the microarchitectural modifications required to support stalling. Section 2.8 provides the related work.

2.2 Background

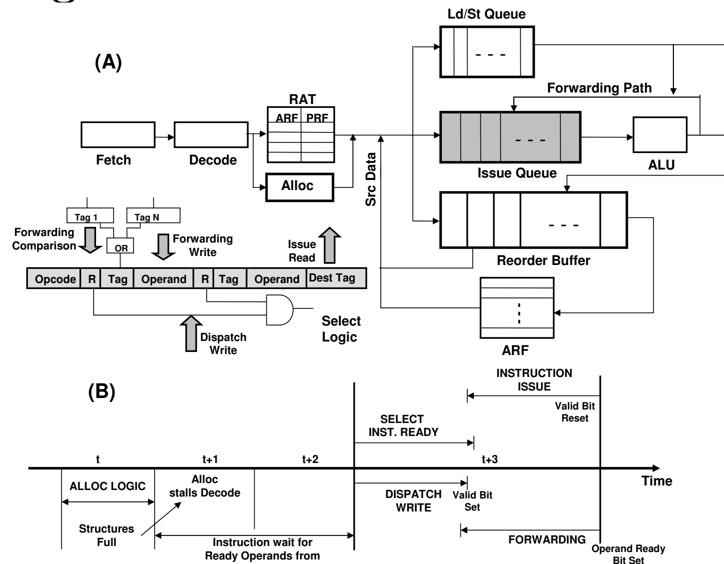


Figure 2.1. (A) Out-of-Order Pipeline highlighting the paths to the Issue Queue. The Figure also shows an Issue Queue entry and all associated activities. (B) Time-line of issue queue activities at cycle-level granularity. Here 't' refers to a cycle. The cycles are not drawn to scale.

Figure 2.1(A) shows the pipeline that we look at in this chapter, similar to those seen in the earlier chapters. Instructions are fetched and decoded in-order in the front-end of the pipeline. Register renaming occurs in the Register Alias Table (RAT), which maintain tags corresponding to the destination architected registers. The Reorder Buffer (ROB) and Physical Register File (PRF) are coupled together and hence the ROB entries correspond to PRF ids. The Architectural Register File (ARF) exists as a separate structure in which instructions retiring from the ROB write their results. Once the instructions get decoded, entries in the issue

queue and ROB are allocated for the instruction in the Alloc stage in parallel with the RAT access. Once an instruction is renamed, it accesses the ARF and ROB for ready operand values before getting written into the issue queue. As soon as the instruction completes execution, it writes its results into the ROB from where they are made available to future instructions until the instruction retires. This ROB access takes two cycles [59].

Issue queues can be dispatch-bound or issue-bound, data-capture or non-data capture style [97]. In this chapter we look at an issue-bound, data-capture style issue queue, where the dispatched instructions fetch their ready operands from the ARF/ROB before moving into the issue queue. A typical issue queue entry looks like the one shown in Figure 2.1(A) [81]. Each individual entry consists of six basic components: the opcode, two source operands, their tags, and the destination tag besides the flags that indicate whether an entry is valid and if its operands are ready. The ready operands, the opcode and the tags get written into the issue queue entries at dispatch. For non-ready operands, their tags are compared against those of instructions forwarding their results each cycle and on a match the operand value gets written into the entry. Once all operands become ready, the instruction issues a request signal to the select logic for it to be selected for execution.

Figure 2.1(B) shows a time-line of activities with respect to the issue queue [44]. The Alloc logic decides whether the decoded instructions can be passed to the back-end based on available entries in ROB, issue queue and load/store queue. Once entries get allocated, the instructions get written into the issue queue. For performance reasons, multiple issue queue activities proceed in a cycle. The dispatch writes of new instructions occurs in the first half of the cycle while forwarding starts in later half. This ordering is important since new instructions could get their source operands from the forwarding path. Once the forwarding match occurs, the operand is written into the entry. Once complete, it sets the operand ready bit. Since forwarded data get broadcast to all entries, the CAM cells of new instructions become effective only after the valid bit is set for these entries. This avoids any meta-stable state in the issue queue entries. Once the operands become ready, the selection and issue of the instruction occur in the next cycle which is extremely important for performance benefits.

Issue queue implementations can be compacting or non-compacting [25]. In compaction-based designs, an instruction issue causes all later entries to move forward. Any new instruction is added only to the tail of the queue implicitly maintaining the oldest-to-youngest instruction order. An alternate implementation is the non-compacting issue queue design, where dispatched instructions are allocated an entry and they remain there until issue. *Holes* created on issue get filled only by newer instructions on dispatch. Past research [25] has proposed selection logic schemes that allow oldest-to-first instruction selection for issue.

In an issue queue, the tags are usually stored in CAM arrays and the data and opcode in SRAM arrays. Prior works have looked into the problems associated with variations in SRAM [4] and CAM cells [10, 76]. Failures in the CAM include search-time, match and SRAM bit failures. Search-time failure is attributed to high V_{th} in transistors discharging the matchline, while match failure occurs when higher leakage currents cause voltage drops on the matchline. SRAM bit failures can be attributed to various conditions, such as data flipping on a read, writes not able to update cells, or V_{th} variations causing data access times to fail.

2.3 Simulation Setup

Baseline Parameters

Parameter	**Value**
Fetch/Decode/Issue/Commit Width	6
Fetch Queue Size	128
Branch-Predictor	Combined Predictor
RAS Size	64
BTB Size	2K-entry 4-way
RUU/LSQ/ISQ Size	128/64/24
Integer ALUs	6 (1-cycle latency)
Integer Multipliers/Dividers	4 (3,20)
FP ALUs	6 (2)
FP Mult./Div./Sqrt.	4 (4,12,24)
1 D-Cache Ports	2
L1 D-Cache	64KB, 2-way 32B block (2)
L1 I-Cache	32KB, 2-way 32B block (2)
L2 Unified	512 KB, 4-way 32B line-size (12)
I-TLB	512-entries 4-way
D-TLB	1K-entries 4-way
TLB Miss-Latency	30 cycles
Memory Latency	150 cycles

Table 2.1. Simulation parameters. Latencies of ALUs/caches are given in parentheses. All ALU operations are pipelined except division and square-root.

Variation analysis of the delays of all our circuits were performed using HSPICE,

a circuit-level simulator. We designed all the required components and performed a delay analysis by statistically varying the device parameters. Our delay simulations employed the Predictive Technology Model (PTM) device models [132] for the 22nm technology. Architectural experiments were conducted using the SimpleScalar 3.0 [24] on the Alpha ISA. The simulator was heavily modified to support the variation-affected issue queue and the consequent stalls induced in the pipeline. The proposed techniques were evaluated on all 26 SPEC CPU2000 benchmarks after fast-forwarding to the single SimPoint [98] and running them for 100 million instructions. The parameters of our baseline model are shown in Table 2.1.

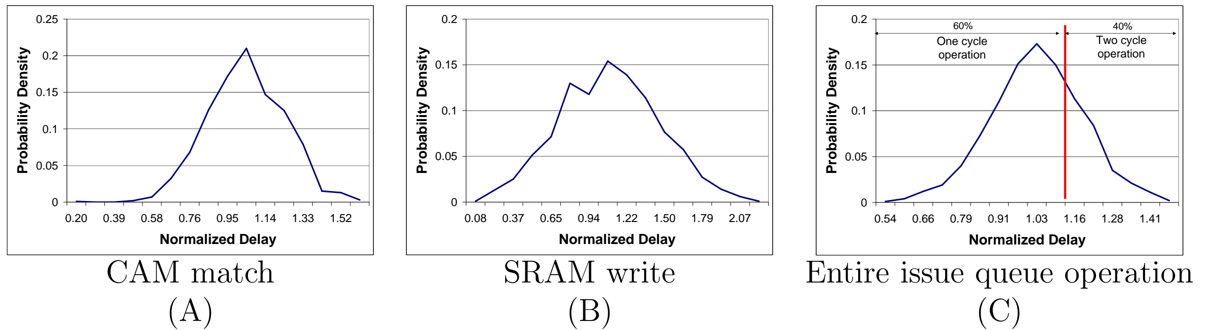
Since PV is a random phenomenon, it could affect any of the entries which have varying impact on performance. Looking at all possible cases would mean analyzing $\binom{TotalEntries}{AffectedEntries}$ cases which would be large. Hence in our simulations we pick one case of PV-affected entries (entries chosen randomly) and study the performance impact across the different schemes. For low IPC benchmarks (IPC < 1.0), their performance is determined by factors like memory accesses (mcf), branches (gcc) or floating point units (equake). In these cases, issue queue variations has less impact on overall performance than in the high IPC benchmarks (IPC > 1.0). Hence the performance graphs give two values, the average values for all benchmarks and the average specific to the high IPC ones, in the legend. All the graphs group the high IPC (gzip to apsi - 15 benchmarks) and low IPC (swim to twolf - 11 benchmarks) together.

2.4 Variation analysis on the key issue queue components

An issue queue consists of CAM and SRAM cells operating together in each entry. Works like [4, 10] have shown timing errors to be more dominant than bit flipping errors for SRAM and CAM cells and hence we analyze them in our work.

A Monte-Carlo analysis was performed for these structures, custom-designed in 22nm technology, by simulating 5000 instances. In this work, we don't model systematic variation as the relatively smaller size of the issue queue, compared to caches in which systematic variations are common, makes random variations the dominant cause of process variation. A variation of 5% in gate length and 10%

in the threshold voltage is assumed. These values were suggested for the 65nm technology by the ITRS and widely used in the literature [4, 67]. The degree of process variation is expected to increase with newer generations [19] and thus our assumptions are conservative for the 22nm technology. Figures 2.2(A) and (B) show the variation in the normalized delay for match operation of a 7-bit CAM cell and write operation of a 64-bit SRAM cell respectively. Figure 2.2(C) shows the variation in the normalized delay of operation of an entire issue queue line. We assume the issue queue to be the frequency-determining stage as indicated by earlier works [81]. A delay of 15% greater than the nominal value (conservative guardband) was chosen to be the threshold beyond which issue queue entries require an extra clock cycle to complete operations. The maximum time required for an operation even in the worst case is within two cycles. Our results indicate the approximately 40% of the entries require the two cycles to complete operation. In our work, this translates to 10 of the 24 issue queue entries.



CAM match (A) | SRAM write (B) | Entire issue queue operation (C)

Figure 2.2. Delay variation in issue queue. (C) shows that 60% of entries operate in 1 cyle while 40% of entries require 2 cycles.

2.5 The effects of PV on issue queue operation

The different pipeline activities that occur every cycle with respect to the issue queue are as follows:

- **Dispatch Writes:** On a dispatch, new instructions are written into their entries. There they wait until they are selected for execution. Dispatching instructions into the issue queue entails CAM writes and SRAM writes.

- **Forwarding Comparison/Write:** When instructions complete execution, they broadcast their results and tags to the issue queue. The dependent instructions use the tag CAM-match logic to identify if their source operands are available in the forwarding paths. Whenever these forwarding comparisons become successful, the operands get written into their entries resulting in a SRAM write.

- **Issue Reads:** When an instruction gets selected for execution, the opcode and operands of the instruction get read (SRAM read operation) from the entry and the instruction is sent for execution.

Each of these activities is important and there is significant interplay between them as well. For example, a slow dispatch write could cause forwarding operations to slow down since the tags do not get written into the entries in time. Similarly if the issue read out of an entry is slow, the destination tags will be read slowly, thus affecting the forwarding paths and any later dependent instruction in the issue queue. To understand the performance impact, we slowed down each of the components and studied the subsequent effect on performance.

Figure 2.3 shows the results of this experiment. We show each of the activities in descending order of performance impact and compare it against the issue queue performance when there is no process variation - NonPV. In certain cases, slow dispatch brings down performance significantly because more instructions are needed to exploit the ILP (mesa, bzip2) while in others the issue queue holds enough instructions to meet IPC requirements and hence slowing down forwarding and issue brings down performance (gzip, sixtrack). Figure 2.3 shows that, across all benchmarks, each of these activities play a vital role in issue queue operation. Hence our variation-tolerant solutions have addressed them together rather than concentrating on one specific activity. Note that slowing down all the activities together, *Slow Dw/Ir/Fw*, has the biggest performance impact, as much as 20%. Our solutions in section 2.6.2 is motivated based on this observation.

2.6 Battling PV artifacts within the issue queue

Our solutions to handle issue-queue variations progressively move from tackling variations at per-entry level to sub-component analysis. Initially we look at steer-

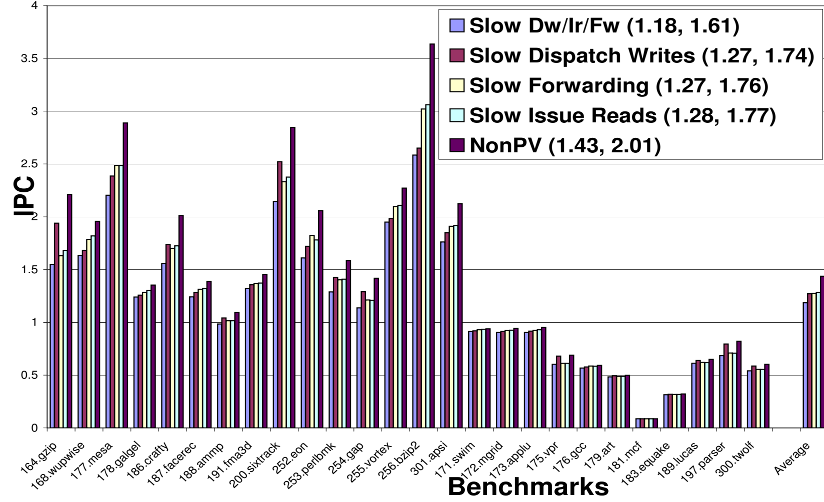


Figure 2.3. Performance Impact of slowing down different activities that occur in the Issue Queue. Legend shows the average IPC across all benchmarks and the high IPC ones.

ing schemes that dispatch instructions to entries based on operating-speeds. Optimizing on these schemes, our intra-entry variation analysis looks at switching operands and ports to further mitigate the impact of variations. Towards the end of the section, we also look at how existing solutions, proposed to handle other phenomena, can be tuned for handling variations.

2.6.1 PV-aware instruction steering

Naive PV-unaware issue queue allocation reduces overall performance by about 20%. Since variation is non-deterministic, entries could either be fast or slow. In this section, we look at steering schemes that optimize entry allocation based on operating speeds of the entries to minimize stalls reducing the performance degradation to 5%. The alloc logic maintains the information regarding the slow entries and steers instructions accordingly.

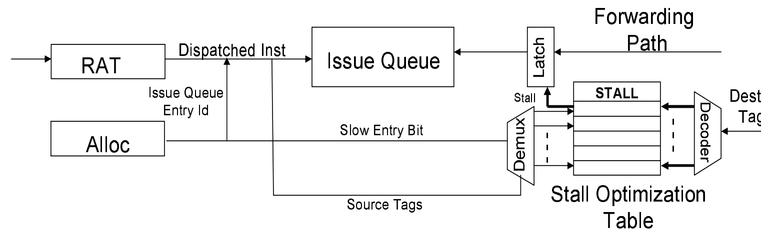


Figure 2.4. Microarchitectural changes required to implement OptiSteer.

2.6.1.1 SpeedSteer: A Speed-aware steering mechanism

Dependent instructions within the issue queue can lie in faster or slower entries. If the instructions lie in slower entries, forwarded results need to be held for an additional cycle in order for slower entries to pick up the data. Since the pipeline does not know if the dependent instructions are lying in faster or slower entries, forwarding can be done every other cycle only bringing down performance greatly.

Our proposed SpeedSteer scheme reduces this by maintaining any dependency chain within faster or slower entries. By doing this, the instructions implicitly know that their dependent instructions also move to similar entry. Hence only instructions in slow entries stall the forwarding paths. When the source operands lie in both type of entries, instructions can be dispatched to faster entries only. In such cases, the forwarding paths might be stalled unwantedly by the instruction in slow entry but this was minimal since the instruction could have other dependents in slow entries.

To identify whether the instruction issued from a fast or slow entry, the instruction carries a bit after issuing. This bit can be added to the entries and set on dispatch – negligible change to variation effects due to adding the bit (size of an entry is 160 bits). Alternatively, the select logic can set this bit while issuing instructions, based on the entries from which they were issued.

2.6.1.2 OptiSteer: An optimized table-based steering mechanism

Simple priority allocation of instructions to fast entries (slow entries used only when fast entries are not available) is not possible with SpeedSteer. This is because there is no mechanism to stall forwarding of instructions that issued from fast entries for an additional cycle when their dependents lie in slow entries. To enable this, OptiSteer uses a table called the Stall Optimization Table (SOT) whose entries are set, when slow dependents are in the issue queue, to stall forwarding.

The SOT has *128 1-bit entries*, corresponding to the destination tags (ROB id) of each instruction. An entry is set if instruction with corresponding source operand tag is in a slow issue queue entry. When an instruction completes execution, it accesses the table to check if forwarding needs to be stalled for next cycle. Figure 2.4 shows how the SOT stalls the forwarding path based on the source operand

tags of instructions. Note that the SOT access is not on the critical path since the value is required not in the current cycle but the next one. Figure 2.13 shows this in greater detail. Given that SOT is a small structure (small access time as well) and its access is not on a critical path, there are no timing issues due to variations affecting it.

Figure 2.5 shows the performance benefits in employing the two instruction steering schemes. The conventional scheme shown as *PV-unAware* is oblivious to variation-affected entries and suffers an overall performance loss of 20.5%. For high IPC benchmarks, this value goes upto 24%. The SpeedSteer scheme in turn exhibits 8.8% (9.2% for high IPC benchmarks) overall performance loss, while the OptiSteer scheme imposes only 5.1% (5.5% for high IPC benchmarks) performance overhead. In certain cases like bzip2, sixtrack and mesa, the PV-unAware scheme leads to 34% performance degradation while the steering schemes have only about 10% performance loss in those cases. The steering schemes do a better job of utilizing the faster entries compared to PV-unAware scheme. Both the steering schemes place about 80% of instructions in fast entries compared to PV-unAware scheme that places only 57% of instructions in the fast entries.

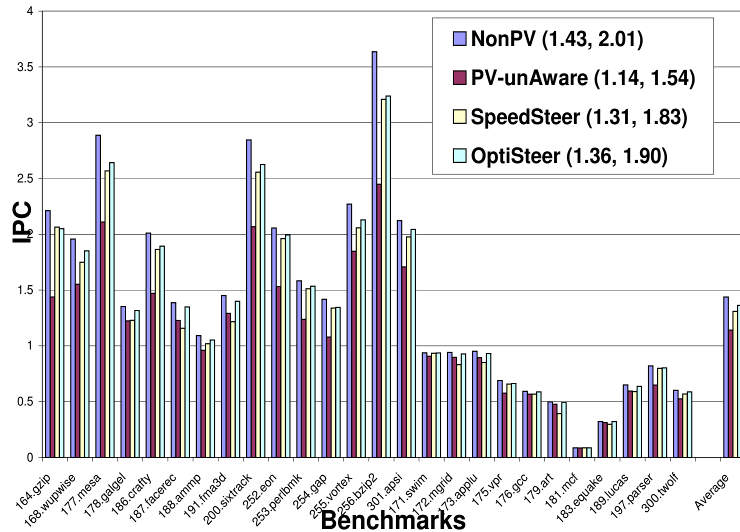


Figure 2.5. Performance obtained by employing the instruction steering schemes. Legend shows the average IPC across all and high IPC benchmarks clearly showing the requirement for the steering schemes to reduce performance loss due to variations.

2.6.2 Intra-Entry variations

Each entry of the issue queue consists of multiple sub-components that differ in their functionality (a tag CAM sub-component and operand SRAM sub-component). Naturally, random variations could cause variability in their behavior. Based on this fact and the observation that all sub-components of an entry need not be useful for every instruction, we provide techniques to avoid variation-affected sub-components whenever possible. Random variability also means that performance of sub-components do not get adversely affected with respect to all the ports used to read or write into the entries. Since the maximum bandwidth is not always required, ports can be switched whenever possible.

Exploiting varying port-speeds would allow entries to have fast dispatch writes while their issue reads could be slow. By identifying this, unwanted stalls could be reduced. The conventional issue queue design allows these operations to be of varying speeds and yet interface with each other. Figure 2.1(B) showing the time-line for conventional issue queue operation which shows that each of the activities when complete sets a flag upon which the next activity begins. CAM cells of new entries are activated only when the valid bit is set and hence slow dispatches in meta-stable state do not participate in CAM match. Similarly a slow forwarding write sets the operand ready bit only after the operation is complete. To handle slow forwarding, the data is held for additional cycle. A slow issue invalidates the valid bit only after it is read and hence new instructions do not get allocated to this entry before that.

2.6.2.1 Operand Switching

Each of the operand sub-components form a significant part of an issue queue entry. In this sub-section, we look at application characteristics that can be used to reduce the variation impact. Based on our observations, we look into several ways of handling variations within an entry.

- **Single Source Operand:** Not every instruction has 2 source operands. If variations affect only one of the operand sub-components, instructions with less than 2 source operands can have their operand (and corresponding tag)

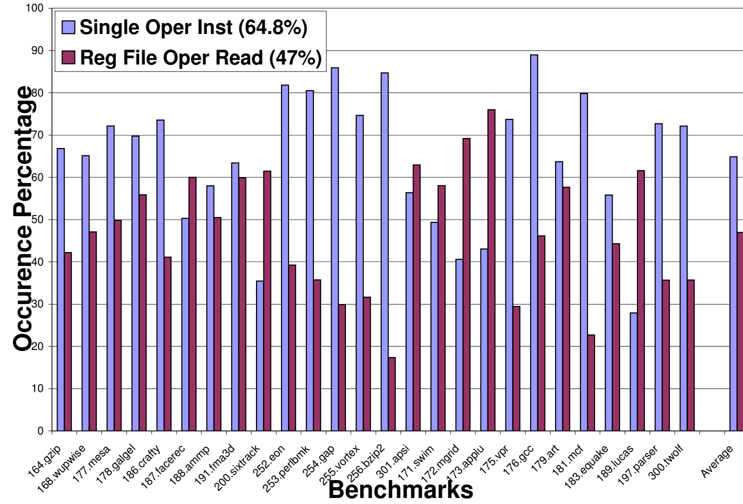


Figure 2.6. Percentage of single-operand instructions and ready operands read out of register file. Legend shows average value across all benchmarks.

steered to the sub-component that operates faster. This avoids the CAM variations in the tag sub-component as well.

- **Operand Ready at Dispatch:** If only one of the operands is ready at dispatch, the ready operand can be steered to a sub-component based on whether it is fast of slow to optimize on the stalls required. For slow dispatch writes to operand sub-component, steering away from it is useful while for slow tag matching/forwarding writes steering to it is useful since the operand is ready.

- **Early Operand Ready:** For two-operand instructions, one of the operands can be ready while the instruction waits for the other operand before issuing. We looked at switching operands (avoid slow forwarding comparisons) based on when they become ready. But the associated complexity in implementation impeded us from using it.

Figure 2.6 shows the percentage of instructions with single operands, 64.8% on average, and the percentage of instructions which read any of its operands out of the ARF at dispatch, 47% on average. Given the percentage of instructions that exhibit these characteristics, it provides a good motivation to switch operands at dispatch.

2.6.2.1.1 Implementation Since the alloc logic knows the specific entry to which the instruction will be dispatched, it also knows whether the entry's operand sub-components are fast or slow. It uses this information to decide whether to switch operands or not. At the end of the renaming stage, we know whether the instruction is of single-operand type and if their operands is ready. Operand-switching is done in the two-cycle interval required to fetch the ready operands from the ARF or ROB. The RAT does not know if the operand in ROB is ready as it only has a bit to look for the operand (in ROB or ARF). Hence we do the operand switching for the operands ready in the ARF.

The operands, switched at dispatch, have to be switched back when issuing to the ALUs. We evaluated the impact of using a transmission-gate-based multiplexer, through a circuit-level simulation, and found that it has an overhead of less than 2% in timing and less than 1% in the total gate count of a single line in an issue queue. To indicate whether the operands within an entry have been switched, a single bit is added to each issue queue entry. Since operand-switching is done at dispatch, the bit is available at issue to control the transmission-gates.

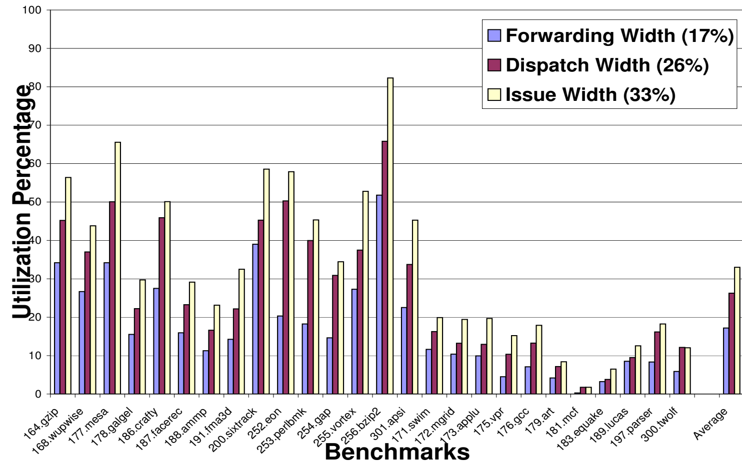


Figure 2.7. Bus utilization for dispatch, issue and forwarding. Average value shows that the effective utilization is low.

2.6.2.2 Port Switching

Port Switching [67] is an effective technique to reduce performance degradation imposed by variations. Random variations lead to variable cell operating speeds with respect to the ports. Hence, dynamic port-switching – whenever possible – can

help reduce the impact of variations. Figure 2.7 shows how the utilization of the dispatch, issue and writeback bandwidth vary across the SPEC2000 benchmarks. Though the utilization is close to half the bandwidth available, port-switching over all available ports might not be possible as discussed below.

2.6.2.2.1 Implementation Even though there are potential advantages to port-switching at dispatch, issue, and forwarding, the specific bus on which the data gets forwarded is decided at issue time by the select logic. The select logic does not know dependents of an instruction. Also, the multiple dependents of an instructions could operate fast or slow with respect to a port since forwarded data get broadcast. Based on these observations we opt not to implement port-switching for data forwarding.

Conventionally, all buses operate at the same speed, and their assignment to instructions is inconsequential. With variations, the bus assignment is non-trivial if stalls are to be minimized.

For dispatch writes, the port assignment can be done in the alloc stage. Since the specific entries that the instruction would be allocated is known, port speeds for those entries would be known as well. This information is used to make the port-assignment. For issue reads, the port speeds of individual entries is main-

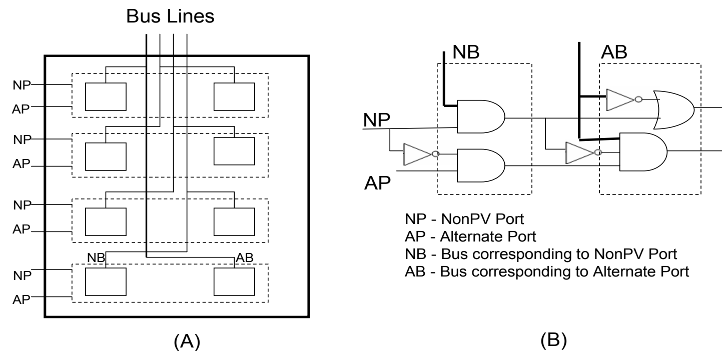


Figure 2.8. Port-switching logic implementation. (A) Overall connections between buses available and port speed lines. (B) Logic inside each of boxes assigning either one of the buses based on availability and port speeds.

tained in the selection logic. Once an instruction is chosen for issue, the selection logic determines the specific port for the instruction. Note that all six ports are not available for the instructions. Since each port interfaces to specific execution

units, port-switching can be done only when instruction can be issued to multiple functional units. Hence we restrict port-switching between two alternate ports. There is one port, *NonPV Port (NP)*, that the instruction would have normally take in a variation-unaffected pipeline. Since this could turn out to be slow, we also check if an alternate faster port, *Alternate Port (AP)*, is available. If it is unassigned to other instructions, the alternate port is taken. Figure 2.8(A) shows the bus and port interconnections required to implement port-switching in parallel, shown for a four-ported system. Figure 2.8(B) shows how the actual assignment is done. Note that a '0' indicates a slow port and a non-available bus while a '1' indicates a faster port and an available bus.

The port-switching logic is on the critical path of instruction issue. To incorporate it, the select logic for non-collapsing issue queue involving four stages to select instructions based on their age, is shrunk to three stages. This reduction in select logic makes it less effective to implement oldest-first selection but port assignment was found to have a more significant impact on performance.

Figure 2.9 shows the impact of combining OptiSteer steering scheme with port and operand switching, shown as *Hybrid ISQ*. Steering is effective since it avoids entries that have predominantly slow ports and hence cause more stalls in pipeline while port and operand-switching are effective when only a lesser percentage of an entry or sub-component are affected by variations. As we note from the figure, the variation affected issue queue now operates within 1.3% of variation-unaffected issue queue. Even for high IPC benchmarks, the loss is only 1.5% clearly showing the capability of our schemes to operate effectively under variations.

2.6.3 Existing mechanisms for handling variations

This section deals with how some of the existing solutions, proposed for other purposes related to the issue queue,can be tweaked for variation handling as well.

Shutting down issue queue entries based on utilization has been traditionally used to maximize the power/energy savings in a system [1]. Since it involves no significant additional effort, entries that violate the timing requirements can be shut down. Figure 2.10 shows the performance impact of such a system. *ISQ_14* indicates the issue queue with 14 entries operating at the desired frequency and rest shutdown. As can be noted from the graph, the performance degradation is

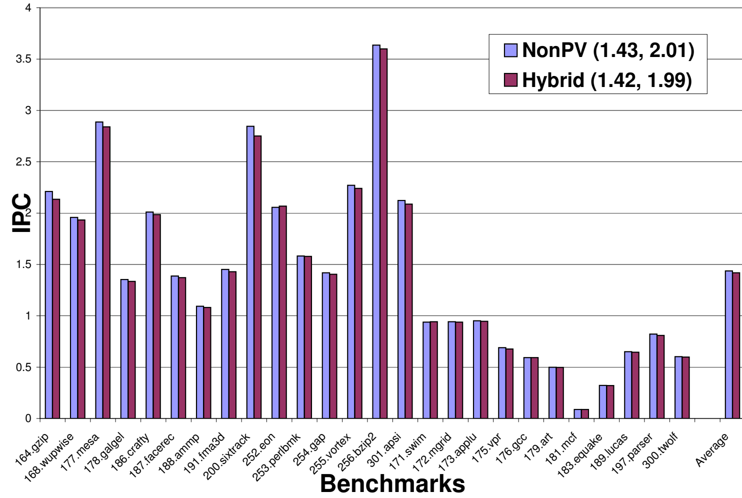


Figure 2.9. Performance impact of combining steering with operand and port-switching. Legend clearly shows that across all benchmarks and high IPC ones, the hybrid scheme does well.

12%, growing upto 16% for bzip2 and sixtrack, compared to the conventional issue queue. Given that variations are only increasing with newer technologies and that in a multi-threading scenario the issue queue's utilization increases, shutdown is not a viable option.

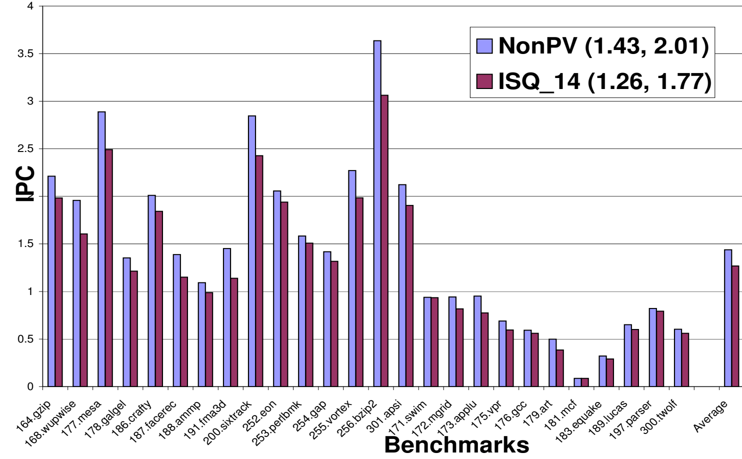


Figure 2.10. Performance Degradation due to shutting down variation-affected entries. Average values across all benchmarks and high IPC ones show that there is significant performance loss with shutting down entries.

Instead of merely shutting down the issue queue entries, an alternate solution we envision is a multi-clock domain (MCD) asynchronous environment. Solutions in the past have looked at operating the issue queue asynchronously with the rest of the pipeline [94] and adapting the issue queue size [1] for reducing the dy-

namic power and/or energy. By operating the issue queue as a fast 14-entry queue or a slow 24-entry queue (incorporating the slow entries as well) and switching between them based on application runtime IPC, we looked at mitigating performance impact of variations. Figure 2.11 illustrates that the MCD-solution leads to performance degradation of about 7.3%.

Compared to both these solutions, our mechanisms are clearly more effective in reducing the performance impact of variations.

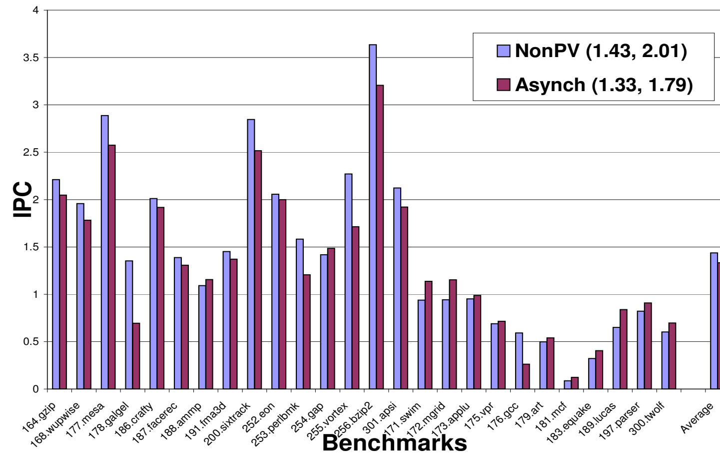


Figure 2.11. Performance obtained by operating the Issue Queue in the two modes in a MCD microarchitecture. The average values show that this might not be an effective solution given the performance loss incurred.

2.7 Epilogue to PV-aware issue queue design

2.7.1 Microarchitectural support for pipeline stalling

Despite all the proposed optimizations for avoiding stalls due to slow entries, there can be scenarios where the pipeline has to be stalled to accommodate the operation of slow entries.

Figure 2.12 demonstrates how stalling is accomplished due to different issue queue activities. What makes the problem under investigation non-trivial is the fact that an instruction could stall the pipeline when (1) its opcode write during dispatch is slow, or (2) its operand read during an issue read is slow. Thus, the stall logic should keep track of these factors.

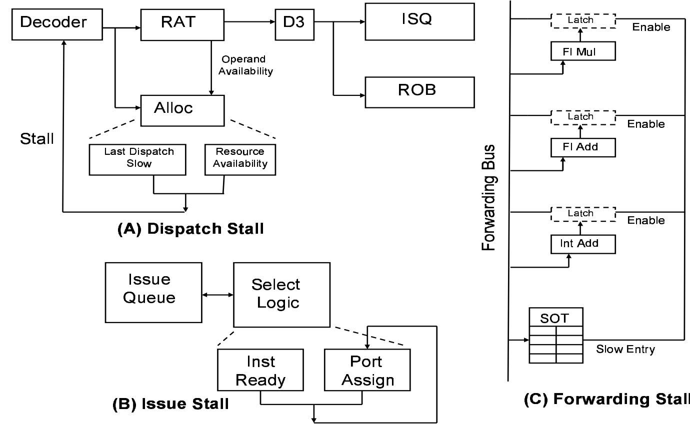


Figure 2.12. Implementation of the stall logic in the different issue queue operations.

For the dispatch stage, we identify whether stalling is needed once renaming is done. Operand availability is passed from the renaming stage to the alloc stage. Since the alloc stage knows the port availability and speed of operation for different entries, it decides whether the next set of instructions need to be stalled. The logic required for stalling the decode stage is already present in a conventional pipeline when entries in ROB, issue queue or load/store queue are not available. In the issue stage, the conventional select logic already keeps track of port availability which it uses for selecting instructions to issue. For slow issue reads, the specific ports would not be available for an additional cycle.

Forwarding stalls are handled using SOT. For handling the stall signals in the execution units, we use the implementation proposed by Hans Jacobson [52], called Elastic Synchronous Pipeline, which uses the ability of master/slave latches to store two distinct values under stall conditions, while storing only one data element under normal operation.

Figure 2.13 shows a time-line incorporating all our mechanisms. The figure clearly indicates that our solutions are mostly not on the critical path and in places they are, appropriate pipeline modifications have been made.

2.7.2 Variation testing methodology

Built in self tests (BIST) aware of variations are gaining importance. Variation-aware testing strategies evolved for identifying slow SRAM entries hold good for identifying slow issue queue entries as well [110, 3]. Also works like [7] look at

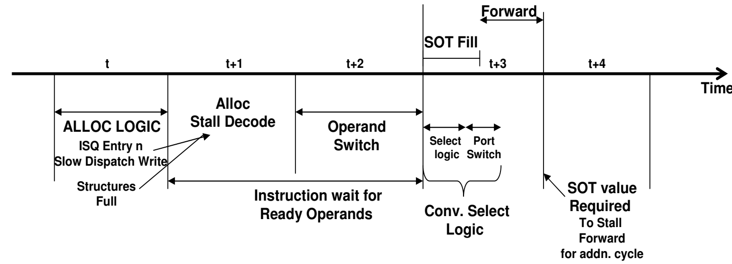


Figure 2.13. Time-line of our mechanisms and stall handling at cycle-level granularity. Here 't' refers to a cycle. The cycles are not drawn to scale.

identifying the target gates that are variation-affected and propose techniques to generate test patterns. These methodologies can be applied for the variation-affected issue queue to obtain maximum performance. Since the different activities associated with the issue queue have varying operation speeds with respect to an entry, this information is maintained at different stages of pipeline for effective operation. Issue queue BIST identifies the 40% of entries that are affected by variations and hence are slow. Using this information in the Alloc stage would enable the steering schemes to operate. Further, by identifying the port-speed information with respect to the different activities we allow instructions to port-switch. Since port-switching is done both at dispatch and issue, the port-speeds are maintained in a ROM in both the Alloc logic and the select logic.

2.8 Related Work

Process variation has been identified as a major hurdle in the coming technology generations. The International Technology Roadmap for Semiconductors (ITRS) has shown a lack of predictability in several aspects of physical design. The intensity of this problem is going to be further aggravated as feature sizes diminish. It has been observed that the loss in performance due to PV can be equal to the gain by one full technology generation [21]. Bernstein [12] presents a survey of process variation issues. Unsal et al. [116] classify process variation based on

source, granularity, manifestation, design parameter and aging. The unpredictability in design due to PV manifests as both random and systematic variations [4]. Lack of predictability in timing characteristics leads to a loss in yield. Strong economic considerations for yield have motivated research at both the circuit and architecture levels to address this problem. PV can also lead to variation in the power consumption of circuits. Borkar et al. [19] show that variation in power consumption can be as high as 20X.

Recent works on process variation have focused on different components such as the register file [67], cache [3] and in redesigning latch elements [38]. Our work is concurrent to [86] which looked at a subset of the issues addressed in this work. An alternate approach is cycle stealing, which allows a PV-affected stage to borrow slack from other stages [111, 67]. While cycle-stealing holds lot of promise in designing variation tolerant circuit, it increases the design complexity and absorbing clock jitter and skew becomes difficult [68]. By removing the variations in issue queue, our work enables the slack to be available for other pipeline stages.

Previous research have looked at breaking the issue queue into multiple structures to make it more scalable [22]. These solutions provide multiple queues operating at different speeds into which instructions are moved based on various conditions. Note that these solutions are built on the fact that the architect has design-time knowledge of the relative operating speeds of entries. The very fact that variations are non-deterministic at design-time makes our problem unique and challenging. Now the fast and slow entries have to co-exist and all the activities must proceed correctly.

Port-switching [67] has been previously studied in the context of register file reads. Here we apply it for the multiple issue queue activities. Prior works [56, 37] have looked at the fact that even though the issue queue is designed to support the worst-case two-operand instructions in all its entries, that might not always be needed. These solutions use their observations to optimize the energy dissipation due to the wakeup logic and hence took a performance hit. Our goal, though, is to reduce the performance impact of the slow entries. Further our techniques take to account the non-determinism due to variations as well.

2.9 Summary

The unrelenting march towards diminutive feature sizes in the deep sub-micron regime has accentuated reliability concerns in modern digital designs. Process Variation is an emerging threat that can adversely affect both performance and power consumption.

The chapter presented the effects of process variation on the issue queue which, to a large extent, determines overall pipeline throughput. Through a detailed analysis of all major sub-components, we identify and quantify the impact of variability on the issue queue to be about 20.5% compared to PV-unaffected issue queue. We demonstrated that solutions targeting individual issue queue operations in isolation are not effective. Hence, we adopt a holistic approach and provided a comprehensive solution that reduced the impact of variations in all pipeline activities associated with the issue queue. The proposed solutions cohesively operate the slower and faster entries of the issue queue in unison, and dynamically optimizes pipeline stalls bringing down the performance degradation to a mere 1.3%.

In future, this analysis need to be extended to other issue queue designs like a collapsible design. Besides the activities associated with the non-collapsible design, collapsible issue queues also have additional movement of instructions between neighboring entries. Variations affecting the data movement logic has a significant impact on performance. This is because there is linear dependency in instruction movement between entries. Variations causing slow movement between any 2 entries would affect all entries after them (slowing them as well), bringing down performance. Since variations is a non-deterministic phenomenon, it is impossible to design the instruction movement logic to minimize impact of variations. The select logic also plays an important role in issue queue operation. In non-collapsing issue queue, the select logic is a linear chain of multiplexers [25]. Hence it is less affected by variations compared to the issue queue itself [21]. Also a slow select logic operation affects only the issue reads. Hence it can be modeled as issue read variations. But given the importance of select logic, it warrants further investigation.

Chapter 3

Mechanisms for Bounding Vulnerabilities of Processor Structures

3.1 Introduction

With the growing need to protect processor structures from transient errors [99, 26], several works have proposed architectural and microarchitectural techniques to provide transient fault tolerance for processor cores. Architecture-level approaches are attractive because of the flexibility they offer in exploring the range of alternatives in the cost vs. performance vs. reliability design space. However, high-level architecture design lies at an early stage in the processor design cycle, and it is difficult to determine at this stage whether an architectural fault-tolerance mechanism will necessarily satisfy the hard reliability budgets that real systems are required to meet. Therefore, architectural fault-tolerance mechanisms must either:

(a) conservatively provide full redundancy to reduce the effective architectural vulnerability of the protected structures to zero while incurring heavy performance or implementation costs, or

(b) provide flexible mechanisms with controllable *knobs*, such that when the design matures and circuit error rates are known, the knobs can be adjusted to

ensure that the mechanism **guarantees** to satisfy any required vulnerability bound.

Previous (micro)architecture-level transient fault tolerance techniques have either implemented full redundancy [91, 117, 74, 82] at the cost of significant performance and/or area overheads, or attempted to explore the performance-reliability tradeoff space, but *without* the ability to guarantee any vulnerability bounds [106, 47, 83, 90].

This chapter presents and evaluates two distinct mechanisms to control vulnerabilities of processor structures and bound them under any specified target budgets. Both mechanisms have fixed implementation (area) costs (which are comparable to previously proposed approaches for full/partial fault coverage), and trade off performance in unique ways to achieve as much fault tolerance as is required to meet the specified vulnerability bound.

3.2 Related Work

A cost-effective technique to achieve perfect fault coverage in the processor pipeline is to exploit the two contexts offered by a simultaneous multithreading architecture to execute multiple copies of the same instruction stream, thereby implementing redundancy [91].

Contention for the shared resources between the two threads has been shown to incur 20%-30% degradation in performance. Addressing this performance loss of SRT, and acknowledging that full redundancy (zero vulnerability) is not a strict requirement for most systems, recent studies [47, 83, 90] have attempted to achieve greater performance by *partial* redundancy mechanisms. For example, opportunistic redundancy [47] uses spare bandwidth in the processor to decide which instructions need to be redundantly executed, with redundancy being avoided during heavy structure utilization periods. Partial redundancy achieves reduced average vulnerabilities compared to single-thread performance, but this approach cannot provide any vulnerability guarantees or bounds. Some other approaches [83, 90] select instructions for redundant execution based on the value and control flow locality in the program. Instructions with unpredictable outcomes/values are redundantly executed, while predictor outputs are used to verify the integrity of the

remaining instructions. These approaches can achieve reasonably small vulnerabilities for processor structures, but yet again are unable to place bounds on these vulnerabilities for any arbitrary application. Further, their utility is necessarily dictated by the value locality in the program. Finally, all these proposals offer one specific point of operation in the performance-reliability design space. It is not clear how much performance can be gained/lost if the system designer mandates a more relaxed/stringent reliability budget.

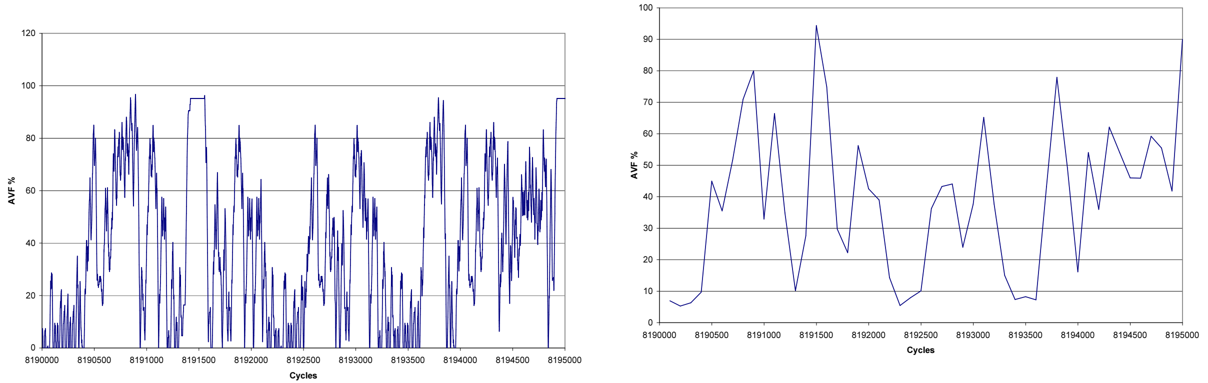


Figure 3.1. Dynamic AVF Variation in *177.mesa* at Cycle (Left) and 100 Cycles (Right) Granularity

Motivation: Instead of exploiting any specific program artifact or performance feature to enhance reliability, mechanisms for vulnerability control of processor structures in order to compulsorily remain within specified reliability budgets are examined. When a system designer specifies a target reliability budget, it is possible that there are periods during the execution when there is sufficient inherent invulnerability, such that no explicit actions need to be taken to enhance reliability. During such periods, high performance can be maintained. On the other hand, there could be periods where the vulnerability exceeds the required bounds, in which case explicit actions need to be taken to enforce the specified constraints. We illustrate these issues by showing the vulnerability (expressed as the Architectural Vulnerability Factor [75], introduced in a later section) of the ROB during the execution of *177.mesa* over a snippet of 5000 cycles in Figure 3.1. The plot reveals that the vulnerability varies quite significantly – reaching near-0% levels at

times, and growing as high as 95% at other instances. If, for example, the designer is satisfied with a vulnerability budget of 20%, then there is a considerable slack during some segments of execution where vulnerability control need not be invoked – to possibly extract higher performance. A key point to be noted is that the granularity of control is also a function of the granularity at which such vulnerability is tracked and needs to be enforced. For instance, the graph on the right hand side of Figure 3.1 shows the AVF for the same period at a granularity of 100 cycles for each sample and demonstrates a much smoother behavior. We assume that the designer mandates vulnerability bounds enforcement at a cycle level granularity.

Our Contributions: Vulnerability control can be specified and performed for any processor structure. Here we primarily apply our techniques to the Reorder Buffer (ROB), a structure that contributes significantly to the processor's real estate.

Our first contribution is in the form of an implementable technique to monitor the architectural vulnerability of the ROB online. Armed with this monitoring infrastructure, we explore two mechanisms for vulnerability control (VC) of the ROB. The first mechanism is a dispatch throttling technique, which estimates the AVF of the structure and proactively forbids an instruction from entering the structure if the action would result in violation of the specified vulnerability bound. The second control technique combines the proactive technique with a reactive mechanism that does not interfere with the flow of instructions, but detects vulnerability bound violations and then reactively performs redundant execution of those instructions which could have caused these violations. In addition, we analyze the effects of committing instructions out-of-order (OoO), which can reduce the residence time of bits in the structure, thereby lowering vulnerability. This reduced vulnerability provides headroom for the vulnerability control mechanisms and enables them to improve performance.

We model and simulate these techniques on a detailed, cycle-accurate processor model, and provide simulation results for all 26 SPEC CPU2000 benchmarks. Our results show that except under extremely relaxed vulnerability bounds, the throttling approach suffers significant performance penalties. Selective redundancy and hybrid approaches on the other hand, provide more attractive performance

levels while ensuring that required vulnerability bounds are satisfied.

3.3 Background and Motivation

3.3.1 Measuring System Vulnerability

The vulnerability of a system component (such as a memory array or a processor structure) is estimated by first performing circuit-level analyzes to arrive at a raw error rate, which can be expressed in terms of Mean Time Between Failures (MTBF) or Failures in Time (FIT) [75]. The raw FIT rate (FIT_{raw}) is then *derated* because certain circuit, microarchitectural and architectural effects reduce the probability that a transient error in the structure will actually lead to an observable error in the output. These probabilities can be encapsulated in terms of *timing vulnerability factors* or TVF (circuit and timing-related effects) and *architectural vulnerability factors* or AVF [75, 65] (architectural and microarchitectural effects). The effective FIT rate of a structure is given by:

$$FIT_{eff} = AVF \times TVF \times FIT_{raw} \tag{3.1}$$

The effective FIT rate of the entire system is obtained by summing up the effective FIT rates of all its constituent components.

3.3.2 Architectural Vulnerability Factors

The AVF of a structure captures the probability that a transient error in the structure will manifest itself in observable output. At a certain point in time, the bits in a structure can be classified as either ACE (required for architecturally correct execution), or un-ACE. Only errors in ACE bits will result in observable errors. Bits could be un-ACE due to several reasons: bits belonging to un-occupied structural entries, bits belonging to mis-speculated instructions or dynamically dead instructions, data bits that get logically masked during computation, etc. (for details, please refer to [75]). The AVF of the structure is defined as the average-over-time of the ratio of ACE bits in the structure to the total number of bits in the structure.

The AVF metric is extremely useful to architects because it de-couples circuit and implementation level effects from architectural and microarchitectural effects on soft error rates, thereby enabling quantitative analysis of architecture-level transient fault-tolerance solutions early in the design cycle of a processor.

3.4 Meeting Reliability Budgets

Vendors need to ensure that their systems meet the requirements of certain *reliability budgets* specified in terms of FIT rates. System reliability budgets can be translated into individual component FIT budgets. Given a target FIT budget for a component or structure, and an estimated circuit error rate, it is possible to determine an AVF bound that an architectural transient fault-tolerance solution must satisfy in order to meet the targeted system budget. However, the circuit-level estimates are usually not known during the high-level architectural design stages of the processor design cycle, thereby mandating architectural solutions to be able to meet any arbitrary AVF bound that can be "dialed-in" when the final circuit estimates become available.

In other words, *if an architectural fault tolerance mechanism is able to adjust itself to meet any specified AVF bound, then this is a sufficient condition that guarantees that this mechanism can be used to satisfy any arbitrary FIT budget for the component.*

3.4.1 Bounding Architectural Vulnerability Factors

We now substantiate the key contributions of this work:

- Given any arbitrary AVF bound for a processor structure, we propose to devise microarchitectural control mechanisms that can guarantee that the AVF bound is satisfied at all times, up to a cycle-level fidelity.

- We present the design and implementation details of two such distinct control mechanisms to bound the AVF of the Re-Order Buffer (ROB) of a processor core – a structure that occupies a significant amount of area in the core's real estate – at any limit specified by the system designer.

  - Our first mechanism is a *proactive* technique that throttles the flow of instructions in order to contain the number of possibly vulnerable bits in the structure within allowable limits.
  - Our second mechanism is a *reactive* approach that allows the instruction flow to execute freely, but tracks instructions that caused the number of vulnerable bits in the structure to exceed the allowable threshold, and selectively executes those instructions redundantly in order to make them invulnerable.

  Both mechanisms have fixed one-time implementation costs in terms of area and implementation overheads, but are flexible in their ability to accommodate any target AVF bound while running any arbitrary application by dynamically trading off as much performance as is necessary to achieve the required reliability.

- We observe that in the presence of such a control mechanism, further AVF-reduction optimizations provide leverage to the infrastructure to boost performance while remaining under the reliability constraints. Long residencies in structures are one of the key contributors to vulnerability, and we observe that retiring instructions Out-of-Order could help to reduce these residencies. We analyze the impact of performing Out-of-Order Commit (OoOC) in a vulnerability-controlled infrastructure.

3.5 Dynamic Vulnerability Monitoring

The first step in building a dynamic vulnerability control system for a processor core is to design an online AVF monitoring infrastructure. This is a non-trivial task, since near-accurate AVF estimation is an exceedingly complex process, even for offline analysis. As described in detail in [75], un-ACEness arises from a variety of conditions. Detection of many of these conditions requires dependency chain analysis across a large sequence of instructions, which is intractable to carry out online. There have been efforts to predictively estimate AVF by observing phased behavior of applications [42], but such statistical estimates are unsuitable

for our purpose since we wish to *guarantee* that the architectural vulnerability of the structure does not exceed a given bound during a fixed interval of time.

3.5.1 Baseline Microarchitecture

We assume a microarchitecture that uses a coupled Re-Order Buffer/Physical Register File, similar to that used in the Intel P6 [97]. Henceforth, we will use the term ROB to refer to this integrated structure. The Architected Register File is maintained as a separate structure into which instructions retiring from the ROB write their results. The Issue Queue stores the operand values (whenever available) along with tags for un-scheduled instructions.

3.5.2 Identifying ACE bits

Although ACE-ness/un-ACEness can be caused by several factors such as dynamic death, logical masking, etc., a significant fraction of the AVF of a structure can be accounted for by tracking the residencies of non-speculative instructions in the structure. Using residency periods alone and ignoring complex dependency-based factors, it is possible to compute an *upper bound* on the AVF of a structure for any length of time. For example, consider a 128-entry ROB (with integrated PRF) in a processor core. Assuming that the Head and Tail pointers of the ROB are invulnerable to faults, if 64 out of the 128 entries in ROB/PRF are un-occupied in a certain cycle, then we can guarantee that the AVF of ROB/PRF can be at most 50% during that cycle. Next, if it is known that the results of certain instructions have not yet arrived in the current cycle, then this factor can be further de-rated. Finally, the contribution of wrong-path instructions to ACE-ness can also be discounted. These are all observable phenomena that can be tracked using simple monitoring logic to provide an upper bound on the ROB AVF at a cycle-accurate granularity.

3.5.3 Online AVF monitoring infrastructure for the ROB

Figure 3.2 shows the logic required to monitor the number of ACE bits in the ROB/PRF of our baseline microarchitecture. The monitoring system is based on

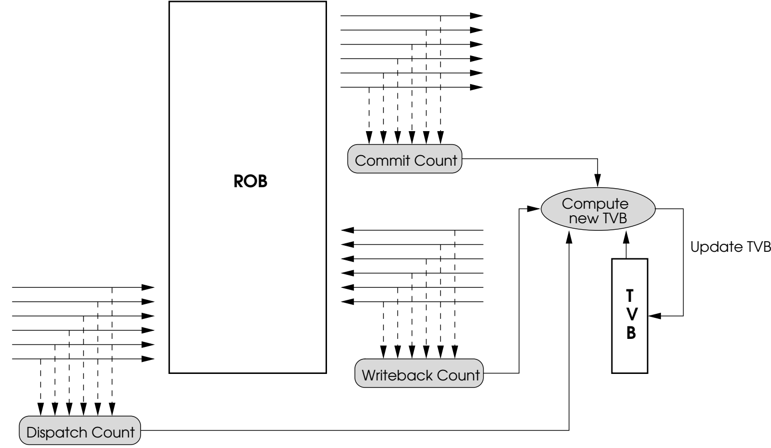


Figure 3.2. Online AVF monitoring infrastructure for the ROB.

the intuition that given the number of ACE bits in the structure in any cycle, it is possible to compute the number of ACE bits in the next cycle by observing the number of dispatches, writebacks and commits into the structure in this cycle. Given that the initial number of ACE bits in the structure when the processor is initialized is zero, it is possible to determine the number of ACE bits in any cycle.

The logic consists of three counters that count the number of (a) Dispatches d into the ROB, (b) Writebacks w into the result field of the ROB, and (c) Commits c out of the ROB every cycle. A register called Total Vulnerable Bits (TVB) maintains the current number of ACE bits in the ROB, and is updated every cycle based on the observed counts. If B is the total number of bits in a ROB entry, and R is the number of bits in the result field, then the TVB for a cycle $i+1$ is given by:

$$TVB_{i+1} = TVB_i + (d_i \times (B - R)) + (w_i \times R) - (c_i \times B) \tag{3.2}$$

The counting logic, to begin with, conservatively assumes that all dispatched instructions are ACE, but when a branch misprediction is detected, the effect of mis-speculated instructions can be computed and accordingly discounted to arrive at the corrected TVB count. Note, however, that only the current TVB count is updated. Ideally, it would have been desirable to retroactively discount the TVB counts of all previous cycles that were impacted by any of the instructions that were just discovered to have been on the wrong path. Unfortunately, this would

require maintaining a history of the cycle when each of the speculative instructions was allocated, which could have significant implementation overheads, and we opt to be conservative. In section 3.7, we show that the selective redundancy control mechanism, in conjunction with this monitoring system, is automatically able to offset some of this conservatism.

3.6 Vulnerability Control via Throttling (VCT)

Throttling is based on the observation that it is possible to bound the AVF of a structure by conservatively controlling access to the structure such that the number of ACE in the structure do not exceed a certain limit at any point in time. We call the number of maximum allowable ACE bits in a structure ACE_{max}. This bound can be computed for the ROB given the following quantities:

$$\begin{aligned} N &= \text{number of ROB entries} \\ B &= \text{number of bits in a ROB entry} \\ R &= \text{number of bits in the result field of a ROB entry} \\ AVF_{max} &= \text{AVF bound (specified as a fraction)} \end{aligned}$$

ACE_{max} for the ROB can then be computed as:

$$ACE_{max} = AVF_{max} \times N \times B \tag{3.3}$$

The monitoring infrastructure gives us an upper bound on the total number of vulnerable bits in the structure every cycle in the form of TVB. Recall that TVB is a conservative upper bound since it is unable to account for the fact that instructions which are on a mis-speculated path, but have not yet been identified as such, do not contribute to the ACE bits in the structure.

The number of ACE bits in the ROB can increase from two events:

- A **dispatch** increases the number of ACE bits by $(B - R)$ (assuming that the instruction is on the correct path).

- A **writeback** increases the number of ACE bits by R.

By comparing the current value of TVB with ACE_{max}, it is possible to determine if a dispatch or writeback would cause the number of ACE bits to exceed ACE_{max}, thereby exceeding the vulnerability bound. In the absence of a *replay-based* scheduling mechanism [57], it is difficult to control or throttle writebacks deterministically. Therefore, we employ **Dispatch Throttling** to bound the vulnerability of the ROB.

Dispatch Throttling is a *proactive* vulnerability control mechanism that stalls instruction dispatch if it determines that dispatching the next instruction into the ROB could cause TVB to exceed ACE_{max}. Given a monitoring infrastructure as described in the previous section, it is very simple to implement. However, it suffers from two key inaccuracies:

- At the stage where the decision to throttle is being made, it is possible that there are speculative instructions in the ROB that are later discovered to be on a wrong path. In order to guarantee AVF bounds, the throttling policy has to assume the "worst case" that all instructions in the ROB are on the correct path.

- The throttling mechanism has no way to determine when the dispatched instruction is going to writeback. It is possible that the $(B - R)$ bits of the instruction do not exceed the vulnerability bound, but the total B bits do, implying that the AVF bound could be exceeded several cycles later when the instruction writes back. Therefore, the throttling policy has to conservatively account for the entire B bits of the instruction while deciding if the instruction will cause a bound violation.

Due to the above two inaccuracies, dispatch throttling has *exactly* the same effect as bounding the total number of entries that can be occupied simultaneously in the ROB. In other words, it is equivalent to virtually reducing the size of the ROB. Nevertheless, the technique is simple to implement, and as our results will show, provides good performance at loose AVF bounds.

3.7 Vulnerability Control via Selective Redundancy (VCSR)

3.7.1 Overview

Selective Redundancy is based on the observation that if an instruction is redundantly executed through the processor pipeline, then the bits in a structure through which both copies of the instruction flowed during their execution are rendered effectively invulnerable. This property is also exploited by *Redundant Multithreading* techniques that fully replicate an execution thread to achieve fault tolerance [91, 88, 74, 82].

Full redundancy results in an effective AVF of zero for all structures within the domain of redundancy (also known as the *Sphere of Replication* or SoR) [91], but suffers from significant performance degradation and is not a requirement for most systems. This has fueled several recent research proposals on partial threading solutions [47, 119, 83, 90] that provide incomplete coverage, but achieve better performance than full redundancy. However, most of these proposals have provided arbitrary single points of operation in the performance-reliability tradeoff space, and it is not clear if any of the operating points have any practical significance.

In contrast, the goal of our VCSR approach is to satisfy a hard vulnerability bound at all times, while attempting to optimize performance within the constraints of the vulnerability bound. We demonstrate a VCSR implementation for the ROB of a modern processor[1].

VCSR is similar in that it monitors the flow of ACE bits through the ROB every cycle and attempts to guarantee that the total number of ACE bits do not exceed a fixed bound. However, unlike VCT's proactive approach, VCSR is a *reactive* technique: In any cycle, if the number of ACE bits exceed the target bound, then a subset of instructions that contributed to the ACE bits in that cycle are selected and redundantly executed, thereby effectively transforming them to un-ACE bits. The advantage of VCSR over VCT is twofold:

- VCSR's reactive nature enables it to avoid a significant amount of the excess

[1] The key principles and mechanisms that we propose should be adaptable to other structures as well.

conservatism of VCT.

- VCSR's cost for redundancy is the over-utilization of processor resources, which has far less performance overheads than the performance reduction due to the stalls and pipeline bubbles caused by VCT.

Like VCT, VCSR is a deterministic technique that *guarantees* that any given AVF bound can be satisfied every cycle.

In the next subsection, we outline the microarchitecture of the basic mechanism required to implement selective redundancy, and subsequently we describe the policy that we use to determine which instructions need to be selected for redundant execution. The monitoring infrastructure, selective redundancy mechanism and instruction selection policy together form VCSR.

3.7.2 Achieving Redundant Execution

In order to support the execution of two redundant threads in the pipeline (although one of the threads only executes a subset of instructions), we start with a baseline Simultaneous Redundant Threading (SRT) microarchitecture. SRT [91] leverages the multiple contexts provided by a Simultaneous Multithreading (SMT) processor [114] to execute redundant copies of the same program. The two execution streams are referred to as the *leading* (primary) and *trailing* (redundant) threads.

The fetch mechanism attempts to maintain a slack between the threads. A Branch Outcome Queue (BOQ) forwards branch outcomes from resolved branches in the leading thread to prevent branch mispredictions in the trailing thread. The leading thread places load values obtained from the data cache into the Load Value Queue (LVQ) for subsequent lookup by the trailing thread. The trailing thread issues loads in program order and absorbs values from the LVQ. At the output comparison point, the address and data of every store instruction are verified by comparing the respective outputs from the two redundant executions. To achieve this, retiring stores are sent out of the store queue into a Store Checking Buffer (SCB) where they wait until the trailing thread catches up. If no discrepancy is detected, a single (architected) copy of the store is retired into the memory system

in program order. Thus, the precise architected state of the machine is defined by the state of the trailing thread.

For VCSR, the leading thread is the primary execution thread and is executed in full. If the AVF bound of the ROB is violated in any cycle, due to excessive number of ACE bits belonging to leading thread instructions [2], then a set of instructions are chosen for redundant execution via the trailing thread. Therefore, a mechanism is required that allows selective redundancy in the trailing thread.

3.7.3 Achieving Selective Redundancy

The primary issue that must be dealt with in order to achieve selective redundancy for the trailing thread is maintaining correct architectural dataflow. Since a partial set of instructions are being executed in this thread, instructions whose producers were not executed require their source operands to be forwarded from the leading thread. Despite this forwarding, redundancy can be maintained if all instructions are checked for correctness before being committed [82, 47].

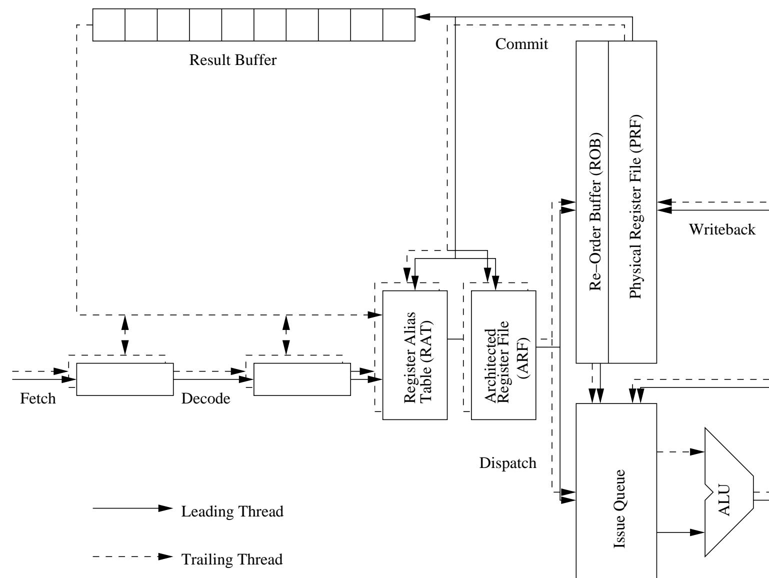


Figure 3.3. VCSR overview.

Figure 3.3 shows a block diagram of our microarchitecture, which is very similar to that employed in [47]. Committing leading thread instructions push their results

[2]In a Single Event Upset model, all bits belonging to the trailing thread are un-ACE.

into a FIFO queue which we call the Result Buffer. The purpose of the Result Buffer is twofold: (a) to temporarily buffer outputs until the trailing thread can compare them against its own outputs for redundancy, and (b) to provide source operands for instructions that the trailing thread does not execute. In addition to the results, the result buffer also contains a single bit that specifies whether the instruction needs to be redundantly executed or not. The logic required to compute this bit is described in the next section.

3.7.3.1 Maintaining a Consistent Architected State

The trailing thread maintains a *consistent architected state* in its Architected Register File (ARF). Recall that the trailing thread does not mis-speculate control flow due to the presence of the BOQ. In VCSR, the trailing thread fetches and pre-decodes all non-speculative instructions regardless of whether they are ultimately chosen for selective re-execution or not. The Result Buffer is examined to determine whether the instruction needs redundant execution. If it does, then the instruction is sent through the rest of the pipeline. The Result Buffer entry needs to be held until the instruction commits so that the redundancy check can be performed. If the instruction does not need to be redundantly executed, then the Result Buffer entry is used to update the trailing thread's ARF entry so that its consumer instructions can obtain the value.

Careful synchronization is needed to ensure that this register update is consistent, as explained below:

- First, the actual update must be performed when this instruction would have been renamed, although it does not physically need to go through the decode and rename logic. This ensures that the RAT is consistent with the instruction flow.

- Second, there could be older in-flight trailing thread instructions in the ROB that write to the same register. Therefore, we use a CAM-style RAT [33] in which committing instructions perform an associative search of ROB/physical register IDs stored in the RAT in order to determine if they were the latest writer. In our case, if we are writing to the architected register from the Result Buffer, this ROB ID in the RAT is set to NULL so that no instruction

already in the ROB will overwrite it. As we will show in Section 3.8, such a RAT is also synergistic with a significant optimization.

We also ensure that all trailing thread Loads, Branches and Stores perform the necessary synchronization with the LVQ, BOQ and SCB so that these structures remain consistent. Since the trailing thread's ARF is maintained in a consistent state, and stores to the system are synchronized with the trailing thread's commit point for the stores, this ARF can be defined as a precise state for establishing checkpoints, both for interrupt handling as well as error recovery.

3.7.4 Selecting Instructions for Redundant Execution

Given an infrastructure to dynamically monitor AVFs and a mechanism to perform selective redundancy, we now need to come up with a policy that selects a set of instructions for redundant execution such that the AVF bounds for the ROB are met every cycle. Ideally, we would like to select the set of instructions that provides the best possible performance among all possible sets that satisfy the vulnerability criteria.

While it is obvious that arriving at such an optimum solution is out of the question for a real implementation, here we provide one implementable *greedy* heuristic.

Recall from section 3.6 that the only two events that can increase the number of ACE bits in the ROB are Dispatch and Writeback. Our VCSR heuristic can flag an instruction for redundant execution at either of the following two stages:

- **During Dispatch:** If it is determined that dispatching an instruction will cause a violation, i.e., if $(B - R) + TVB > ACE_{max}$, then the instruction needs to be marked as requiring redundant execution. This flag is stored in the ROB until commit, upon which it is transferred to the Result Buffer.

- **During Writeback:** If a writeback is causing a violation, i.e., $R + TVB > ACE_{max}$, then the instruction needs to be flagged for redundant execution.

The above description is a simplified view of the process. In reality, several instructions could be undergoing dispatch, writeback and commit simultaneously in a single cycle, and the flagging mechanism needs to take all of these as input to

determine the set of instructions to be flagged. We give priority to younger instructions for removal since they have a greater probability of being mis-speculated instructions (see discussion below).

The detection and flagging logic can be kept off the critical paths of the dispatch and writeback logic by using independent ports to the access the flag bit in the ROB, and allowing for an additional cycle to write the flag bit if necessary. This is not unusual since the different fields in a ROB are typically accessed via independent ports in real microprocessor implementations.

Note that an instruction selection logic could have potentially chosen from amongst any of the instructions currently residing in the ROB, and not necessarily from amongst the instructions currently undergoing dispatch or writeback. However, selecting in this manner simplifies the decision logic, maximizes selection of possibly mis-speculated instructions (see below), and allows the flag bit to potentially share write ports and address decoders with the status bits.

3.7.4.1 Impact of Mis-Speculation

We count ACE bits from speculative instructions as part of TVB. This means that the actual number of effective ACE bits in the ROB could be less than TVB, if there are mis-speculated instructions in the structure. This leads to the concern that our heuristic could be overly conservative in selecting instructions for redundant execution, and therefore perform worse than what would have been possible had the decision to re-execute been delayed until a later stage in the pipeline. Fortunately, if there are any mis-speculated instructions in the ROB, then *it is guaranteed that any instructions that are waiting for dispatch are also on the mis-speculated path*. Since mis-speculated instructions are never executed by the trailing thread, there is no unnecessary re-execution.

However, it is not possible to guarantee that instructions flagged during writeback will not lead to over-conservative re-execution. To minimize this occurrence, we try to select the youngest among the currently writing-back instructions.

3.7.5 Hybrid Vulnerability Control (VCH)

A VCSR-capable infrastructure can be augmented with a dispatch throttling mechanism to combine the advantages of both approaches. In theory, it is possible to devise a multitude of heuristics to determine which control mechanism to exercise based on observed conditions. We show that a simple VCH policy can help to improve the performance of a baseline VCSR system by a reasonable margin.

Recall that one of the key problems with VCT was its over-conservatism due to its inability to discount mis-speculated instructions and un-ACE bits arising from empty result fields for instructions that have not completed. In our hybrid approach, we perform dispatch throttling aggressively by counting only the $(B - R)$ bits for instructions in the ROB that have not yet written back and for new instructions being dispatched. VCSR is used as the fallback mechanism if bounds are exceeded. Recall that VCSR reduces the effect of mis-speculated instructions by giving priority for youngest dispatched instructions. Only when every instruction that might get dispatched this cycle will end up being replayed does throttling becomes active. It prevents the leading thread from being dispatched but makes the entire dispatch bandwidth available to the trailing thread.

3.8 Optimizations

Apart from transient fault tolerance techniques that have moderate implementation and/or performance overheads such as redundant execution, it is also possible to employ lightweight, simple-to-implement AVF-reduction techniques for certain microarchitectural structures. Most of these techniques address the fact that vulnerability arises from long residency of inactive bits in structures, and attempt to reduce such long residence periods. For example, flushing a cache periodically has been observed to produce moderate reductions in AVF while incurring little to no performance loss [122]. Similarly, flushing the processor pipeline on experiencing an L2 miss can also reduce the AVF of processor structures [47].

The vulnerability control mechanisms we have proposed in this work are capable of exploiting the benefits offered by such AVF reduction optimizations. By reducing the effective AVF of structures, these optimizations provide more headroom for our

control mechanisms to enhance performance.

In this section, we propose to reduce the AVF of the ROB by a novel adaptation of a previously-proposed concept – Out Of Order Commit (OoOC)and analyze its behavior when used in conjunction with VCT and VCSR.

3.8.1 Out of Order Commit

Instructions need to commit in-order out of the processor pipeline to provide the illusion of program-ordering to the programmer. This is critical for precise handling of exceptions and interrupts. Unfortunately, it also leads to inefficiencies in the ROB. Since execution of instructions is performed out-of-order by a modern superscalar processor, younger instructions that have completed need to wait in the ROB until they reach the head of the structure.

A number of research works in the past have attempted to tackle this problem and devise solutions that allow instructions to commit out-of-order and yet maintain correctness for handling interrupts and exceptions [9]. Within the context of our vulnerability control infrastructure, OoOC is interesting because of three key reasons:

- In-Order Commit can cause significant Complete-to-Commit delays that play a large role in increasing the AVF of the ROB. Reducing this Complete-to-Commit delay has the potential to reduce AVFs and thereby improve performance with the aid of a VC mechanism.

- One of the key implementation issues with OoOC has been that of re-claiming the ROB entries from instructions that have committed out-of-order, without which there may not be any performance benefits. However, with VC, we observe that even if entries are not reclaimed, simply retiring instructions out of order renders the entries occupied by these instructions un-ACE and reduces ROB vulnerability, thereby improving performance.

- In VCSR, since it is the *trailing thread* that maintains the precise architected state of the machine, the leading thread can be committed out of order without having to provide any additional infrastructure to handle precise exceptions, thereby making a strong case for an implementable OoOC

mechanism.

3.8.2 OoOC and AVF reduction

The architected register file (ARF) of most processors are typically 8-32 entry RAM structures that are significantly simpler than the large, multi-ported, 128-entry ROBs and Physical Register Files. Therefore, ECC or parity protecting an ARF is relatively cost-efficient to accomplish and simplifies the implementation of several fault-tolerance mechanisms [88, 82]. Therefore, early removal of data from an un-protected ROB/PRF into a protected ARF results in AVF reduction for the ROB without affecting the reliability of any other structure.

When used with VCSR, even the ARF need not be protected. Retiring a leading thread instruction from the ROB in VCSR requires writing the result to both the leading thread's ARF as well as the Result Buffer. This act of replication implicitly provides redundancy to the data element. AVF of the ROB AVF is reduced, but neither the ARF nor the Result Buffer need to be protected.

3.8.3 Ensuring Correct ARF updates with OoOC

If instructions are retiring out-of-order from the ROB, there needs to be a mechanism that makes sure that older instructions do not overwrite results that were previously written into the ARF by younger instructions. This can be achieved by using a CAM-style RAT [33], where the RAT stores the ROB tag for the youngest in-flight instruction that owns every architected register. Retiring instructions associatively search the RAT (easily accomplished since RATs have 8-32 entries), and if a tag match is not found, no update occurs.

3.8.4 OoOC without Collapsing (OoOC-NC)c

In our OoOC approach, we allow instructions to commit out of order, but do not attempt to reclaim or collapse entries freed in the middle of the queue. Although the freed entries cannot be used for incoming instructions, the number of ACE bits in the ROB is reduced, thereby providing additional headroom to both VCT and VCSR to achieve higher performance. There is no requirement of any additional

ids to be maintained in ROB and more importantly avoid the case of associatively searching the ROB.

In-Order Commit requires the retiring logic to monitor only the oldest c (commit width) entries in the ROB to determine which instructions are ready to commit. However, for OoOC-NC, the entire ROB needs to be monitored each cycle to determine the set of instructions ready to retire. This can be accomplished by using a priority-based selection tree. The wire delay for such a logic tree is of the order of $log(n)$ where n is the number of entries in the ROB [80].

3.9 Results

Our experiments were conducted via execution-driven simulation using a processor models that we implemented using the SimpleScalar 3.0 toolset [24]. We evaluated our techniques using all 26 applications from the SPEC CPU2000 benchmark suite. The benchmarks were compiled for the Alpha ISA and use the reference input set. We measured the statistics for detailed simulation of 100 million instructions after fast-forwarding to the single SimPoint [98] of each benchmark. The parameters of our baseline model are shown in Table 6.1.

Baseline Datapath Parameters

Parameter	Value
Pipeline Width	6
Pipeline Stages	15
Fetch Queue Size	16
Load Value Queue (LVQ) Size	128
Branch Outcome Queue (BOQ) Size	128
Store Checking Buffer (SCB)	64
Branch-Predictor	Combined predictor with 16K-entry meta-table
RAS Size	64
BTB Size	2K-entry 4-way
RUU Size	128
LSQ Size	64
Integer ALUs	4 (1-cycle latency)
Integer Multipliers/Dividers	2 (3,20)
FP ALUs	2 (2)
FP Mult./Div./Sqrt.	1 (4,12,24)

VCT/VCSR Parameters

Parameter	Value
AVF bounds	0%, 2%, 5%, 10%, 20% 50%, 100%
Result Buffer	128-entry RAM segment + 384-entry FIFO segment

Memory System Parameters

Parameter	Value
L1 D-Cache Ports	2
L1 D-Cache	64KB, 4-way, 32B block (2)
L1 I-Cache	64KB, 4-way, 32B block (2)
L2 Unified Cache	512 KB, 4-way, 64B line-size (12)
I-TLB	512-entries 4-way SA
D-TLB	1K-entries 4-way SA
TLB Miss-Latency	30 cycles
Memory Latency	200 cycles

Table 3.1. Simulation parameters. Latencies of ALUs/caches are given in parenthesis. All ALU operations are pipelined except division and square-root.

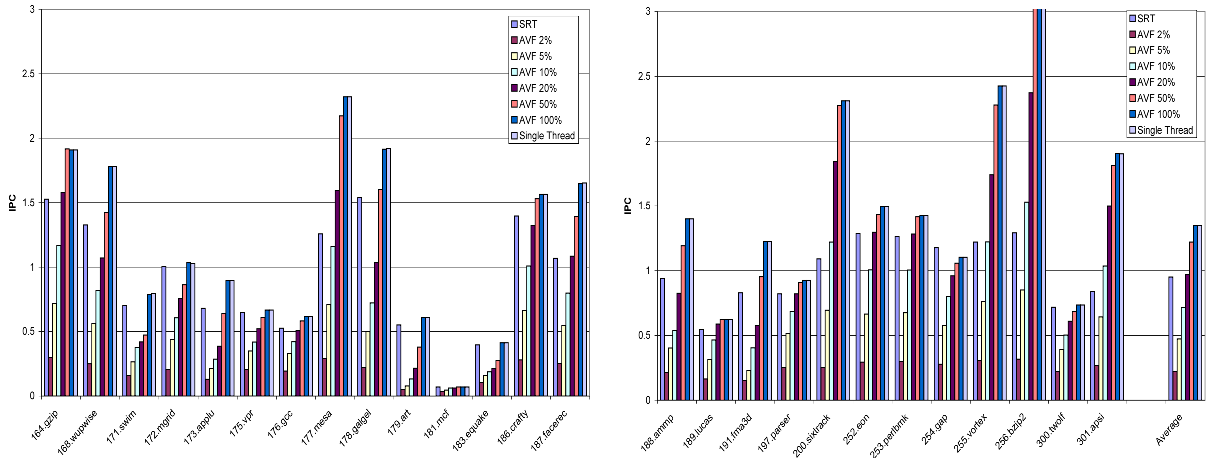


Figure 3.4. IPC with VCT for various AVF bounds. Also provided for each benchmark are IPCs for single-thread execution (rightmost bar) and SRT (leftmost bar).

3.9.1 VCT Results

Our first set of results is for vulnerability control with throttling (VCT), and in Figure 3.4 we give the performance of this technique for a wide range of AVF bounds, together with the performance of a single threaded execution (which is completely vulnerable) and the baseline Simultaneous Redundant Threading (SRT) system [91] (which provides complete redundancy). Note that in VCT, an execution for AVF bound of 0% does not make sense, since no instructions will be dispatched. Consequently, the IPC of VCT with 2% AVF bound is compared with the SRT execution. At the other end, VCT with 100% AVF bound does not throttle any instruction, and is thus equivalent to single-thread execution.

We see that a simple throttling mechanism is able to provide a wide-spectrum of operating points in the performance-reliability space. This range of operation can be better understood by examining the AVF during each cycle of the execution. Figure 3.5 plots the Cumulative Density Function (CDF) of the AVF during each cycle for five representative applications (rest are omitted for clarity) in the single threaded execution. Based on these results, we make the following observations:

- This mechanism is effective only when one is willing to tolerate a high level of AVF (typically above 50%). On the average, there is only a 9.3% drop in IPC when going to a 50% bound from a 100% bound, while there is a 83% drop

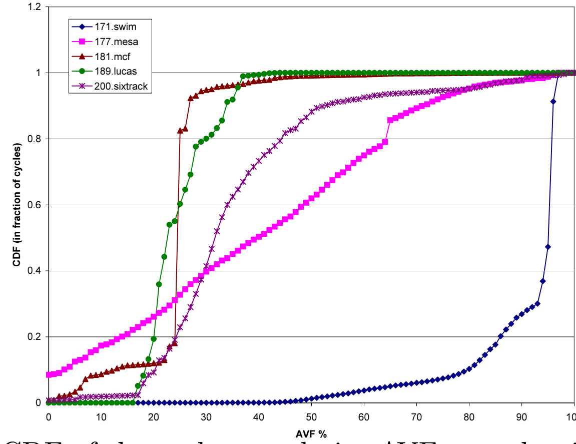


Figure 3.5. CDF of the cycle granularity AVF over the single-thread execution.

when going down all the way to 2%. At these high AVF values, there is not a significant impact from slightly under-utilizing the ROB. This is particularly true in applications where the ROB is anyway not operating at full capacity. For instance, in *200.sixtrack*, where the ROB occupancy in the single-thread execution is 51.6% on the average, there is very little change in IPC going from 50% to 100% AVF. The low occupancy of the ROB is also reflected in the AVF CDF graph for *200.sixtrack* (where the number of cycles with AVF greater than 50% is quite low), as is the case for *181.mcf* and *189.lucas*.

- At the other end, going down to a small AVF bound severely throttles the pipeline, restricting the exploitable parallelism. The penalty is particularly acute for high IPC applications (e.g. *256.bzip2*, *177.mesa*, *200.sixtrack*). This restriction leads to a very poor IPC, if one wants small AVF bounds, making it a much inferior alternative to a mechanism such as SRT which provides complete redundancy. One needs to tolerate an AVF bound of 20% or higher, on the average, in order for VCT to become competitive with SRT.

- The steepness of the IPC improvement with increasing AVF bounds can be explained with the AVF graphs shown in Figure 3.5. (i) In applications such as *177.mesa*, the CDF graph shows that the AVF of the ROB in each cycle is more or less evenly distributed between 0-60%, implying that a 40% AVF bound will throttling half as much as a 20% AVF bound. This effect can be seen in the steady IPC improvement when going from 2 to 50% AVF bound.

(ii) In applications such as *181.mcf* and *189.lucas*, the AVF CDF graph shows high convexity, i.e. most cycles have AVF less than 40%. Consequently, the steepest performance losses in VCT is incurred for AVF bounds smaller than 20%. (iii) At the other end, *171.swim* is an example application with a concave CDF graph, where most cycles have very high AVF values (close to 100%). The resulting IPC graph shows that there is a significant gap in performance between 50 and 100% AVF bounds for this application.

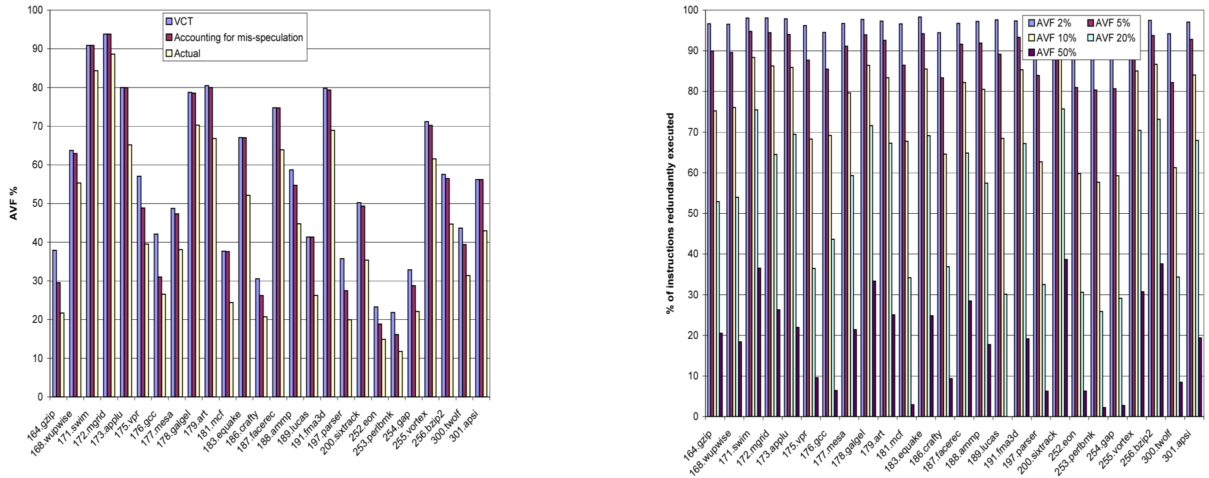


Figure 3.6. (A) AVF Estimation of the ROB by VCT averaged over all the cycles highlighting the conservatism in throttling. (B) Percentage of instructions redundantly executed with VCSR for different AVF bounds.

Another factor which makes VCT a less attractive vulnerability control mechanism is the possible overestimation of vulnerable bits of the ROB in any cycle. As explained earlier, we are assuming that the output result bits of an instruction are vulnerable as soon as it is dispatched. Further, we are also accounting for the vulnerability of mis-speculated instructions. We quantify this conservative estimation of vulnerable bits by VCT in Figure 3.6 (a), by comparing its average AVF with that for (i) an execution which accounts for the output result only when they become available, and (ii) an execution which accounts for the output result bits as well as mis-speculated instructions. As can be seen, across the applications, there is an absolute error of around 10-20% in the AVF estimation by VCT (most of it contributed my mis-speculated instructions). While one would think that

the performance degradation of VCT will be more prominent in applications with high AVFs, the error in its estimation can cause VCT to perform poorly even in applications with low AVFs. For instance, even though *253.perlbmk* has an average AVF of close to 10%, we see that a 20% AVF bound VCT execution performing poorly because the conservatism puts the estimation over 20% on the average. Applications with lower AVFs are thus hurt more by the over-estimation in VCT.

3.9.2 VCSR Results

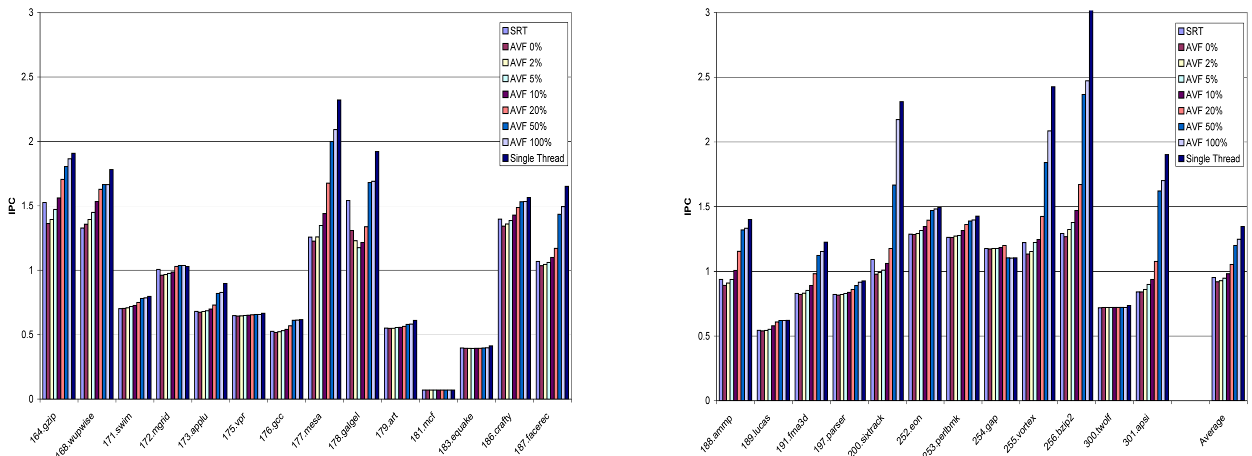


Figure 3.7. IPC with VCSR for various AVF bounds. Also provided for each benchmark are IPCs for single-thread execution (rightmost bar) and SRT (leftmost bar).

Figure 3.7 shows the VCSR results for different AVF bounds, in comparison with SRT and single-thread executions. A good VCSR implementation should perform close to single-threaded execution for a 100% AVF bound (i.e. it does not pick any instructions for redundant execution), and close to SRT for a 0% AVF bound (i.e. all instructions are selected). In these two scenarios, our VCSR implementation suffers a performance degradation of 7% and 3% respectively, which we consider to be fairly efficient.

As in the earlier set of experiments, lowering the AVF bound reduces the IPC of the execution. In VCSR, this is because more instructions are picked for redundant execution as is shown in Figure 3.6 (b). However, the steepness of the decline in IPC for low AVF bounds is much less than what we noticed in the case of VCT. This can be explained by two main advantages that VCSR has over VCT. First,

VCSR is reactive and takes appropriate actions only when the AVF has *actually* exceeded the bound, compared to VCT which takes pro-active actions based on predicted (over-estimated) AVF values. Second, stalling based control can have more detrimental performance consequences (reducing parallelism, introduce bubbles, extend critical paths, etc.) on pipeline performance in VCT, compared to higher resource contention between competing threads in VCSR. The latter artifact is particularly apparent in memory bound applications such as *171.swim* and *189.lucas*, where memory stalls in the leading thread can cause less contention for datapath resources between the leading and redundant executions. Consequently, even when over 95% of the instructions are redundantly executed, the performance loss compared to single-thread execution is only around 5% in these applications. This is also the reason why SRT performance in these applications is not significantly worse than single-thread execution either.

Resource contention in VCSR is expected to play a more detrimental role in applications with high IPC. For instance, applications such as *256.bzip2*, *255.vortex* and *200.sixtrack*, show significant drop in IPC even when going to AVFs of 20% with VCSR. In some of these cases, VCSR actually does worse than VCT for high AVF bounds. For instance, *200.sixtrack* with a AVF bound of 50% does 27% worse in VCSR compared to VCT. The resource contention in these high ILP applications is more detrimental, and throttling to slightly lower the effective ROB size is a more effective option to keep the AVF under control when the bounds are lenient.

3.9.3 VCH Results

Summarizing the results from the previous two sets of experiments, we see that (i) pro-active throttling (VCT) hurts when the AVF bounds are stringent (i.e. the drop in IPC gets steeper for low AVF bounds), and (ii) reactive selective replication (VCSR) causes resource contention for high ILP applications when the AVF bounds are loose (i.e. the drop in IPC is steeper at high AVF bounds than at low AVF bounds) and incurs overheads in its implementation (i.e. at 100% AVF bound, the performance is still not the same as single-thread). We pictorially illustrate these observations in Figure 3.8. This graph plots the performance normalized

with respect to single-thread execution for different AVF bounds. Rather than show this for each application, the graph has been drawn by taking the geometric mean of the slowdowns observed by all 26 applications. As we can see, the curve for VCT depicts worse performance at stringent AVF bounds, but crosses over the curve for VCSR for AVF bounds greater than 75%.

Of these two, VCSR is certainly a much better option because it offers meaningful operating points for a wide range of useful AVF bounds, including an AVF bound of 0% (i.e. *VCSR subsumes SRT*) which is not offered by VCT. However, it is possible to integrate these two mechanisms to co-exist, as was explained earlier in section 3.7.5. The line, VCH, in Figure 3.8 plots the performance of this hybrid mechanism. We see that VCH does better than the other two for a wide spectrum of specified AVF bounds. Even for an AVF bound of 2% it brings the performance within 18% of single thread IPC on the average, and it is 10% better than SRT.

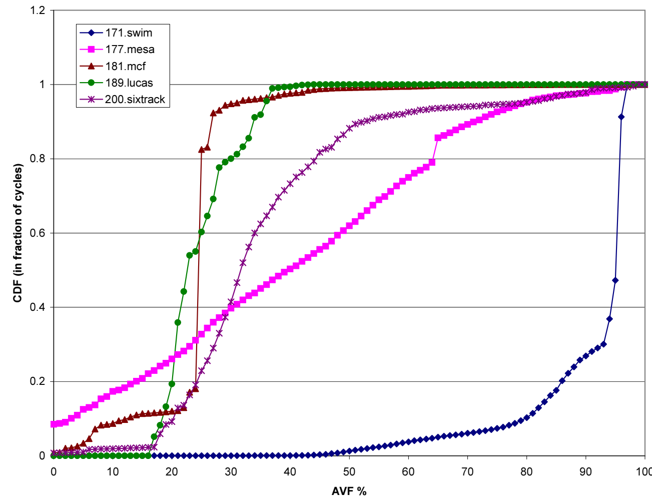


Figure 3.8. Summary of Results. Geometric Mean of IPC across Applications Normalized w.r.t. Single Thread IPC for different AVF bounds. Also shown is SRT performance on the y-axis (i.e. AVF=0).

3.9.4 Non-Collapsing OoO Commit

Until now, all the experiments retired entries from the ROB in-order. Our final set of results are intended to show how committing instructions OoO - to reduce vulnerable bits in the ROB - can help our control mechanisms boost performance.

In the interest of clarity, we show this for only the VCSR control mechanism in Figure 3.9. Note that these results are with a non-collapsing ROB, where the vacated entries are left empty until they reach the head of the ROB. Consequently, SRT and Single-Thread performance is not going to be any different from that shown earlier in Figure 3.7, and is not repeated in these graphs.

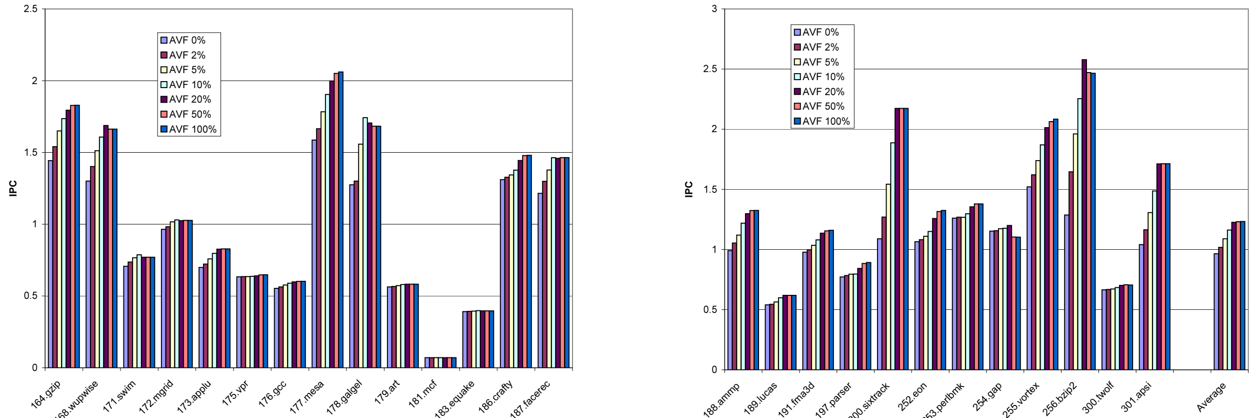


Figure 3.9. IPC with VCSR using Non-collapsing OoO Commit

By reducing residency times of instructions in the ROB, the VCSR control mechanism finds greater vulnerability slack, and needs to step in less often to mark instructions for redundant execution. The consequent reduced number of redundant instructions automatically boosts the performance as we can see in these results. This improvement is particularly noticeable in high IPC applications where resource contention due to redundant execution with a low AVF bound was hurting performance previously. For instance, when we consider *200.sixtrack*, we now see that the IPCs with AVF bounds of 20% and 50% matching those for a bound of 100%, while the IPC of the 20% bound was half as much in the in-order execution. We have also conducted experiments for OoO commit with VCH, and the resulting performance is summarized in the line shown as “VCH(OoO)” in Figure 3.8. As can be seen, the benefits of removing vulnerable bits are apparent here as well, further improving the performance over a wide range of AVF bounds.

This exercise points out that techniques for vulnerability reduction - such as OoO Commit or even the previously proposed pipeline flush upon L2 miss [47] - can complement our control mechanisms towards providing greater vulnerability slack for boosting performance.

3.10 Summary

This chapter presented knobs for controlling the vulnerability of processor structures, that can be easily modulated to meet target reliability budgets specified by system designers. These budgets are treated as hard constraints, and we have proposed two main knobs - dispatch throttling and selective redundancy - to control the vulnerability of the ROB. We have designed simple hardware implementations to monitor the vulnerability of this structure, which is used by the control mechanisms proactively and/or reactively to ensure that the number of vulnerable bits does not exceed the specified limits. Using detailed simulations we have evaluated the pros and cons of these approaches, and compared them with executions that perform complete redundancy and those which are fully vulnerable.

Our results show that throttling suffers severe performance ramifications when the specified budgets are stringent, but does have low overheads helping this scheme over the selective redundancy approach when the budgets are very relaxed. It is also rather straightforward to implement. The selective redundancy mechanism, on the other hand, can span the entire performance spectrum between complete redundancy and single thread execution while meeting the specified budgets. We have also seen the benefits of integrating these two mechanisms to have a wide range of performance efficient operation. Further, this chapter proposed a novel adoption of Out-of-Order instruction commit, which does not have many of the usual complications in its implementation. We do not need to reclaim the vacated holes left by committed instructions in the ROB, since our goal is to only reduce the vulnerable bits. By reducing the vulnerability, our control mechanisms have higher AVF headroom to instead boost performance.

One issue that needs further consideration is the ramifications of relaxing the specifications of vulnerability bounds so that we need to adhere to them only with a high percentile rather than in every cycle.

Chapter 4

Multicore Soft Error Vulnerability Characterization

4.1 Introduction

With the continuing increase in on-chip transistors, multicores seem to be the only cost-effective alternative to address the power and design constraints imposed by the deeper levels of transistor integration. Multicores allow the exploitation of thread-level parallelism (TLP), as opposed to the traditional instruction-level parallelism (ILP) on single cores that have several limitations, to continue meeting the performance demands of applications. Consequently, Chip-Multiprocessors (CMPs) or multicores (used interchangeably), have become the de-facto platform of choice from the low power embedded market to the high-end datacenter and supercomputing domains. We are already seeing 2 to 8 core CMPs in the market [54, 58, 50] and this number will only increase in the future [35].

A side-effect of the increasing transistor count is the requirement to lower operating voltages for keeping total system power within bounds. However, lower voltages make transistors more prone to permanent and transient faults. As discussed in chapter 3, the exponential increase in on-chip transistor count has led to a corresponding increase in the overall system SER [17, 73]. Growing transistor and core counts, and the continuing push for high performance - whether it be for a single application or for the throughput needs to accommodate higher degrees of

server consolidation in a virtualized setting [49] - makes the problem of providing low overhead solutions to mitigate soft errors extremely important.

Analyzing the vulnerability of these multicore platforms to soft errors is the first step to identifying, developing and implementing protection mechanisms (in hardware or software) that usually involve tradeoffs between the level of hardening that can be provided and the associated overheads (in terms of performance, power and cost) in providing this hardening. While such analysis exist for uniprocessors [118], these are not directly extendable for multicores running multi-threaded applications since the dependencies (data and synchronization) between the threads and the idiosyncrasies (scheduling on/across the cores) can have additional effects as will be shown in this chapter. To our knowledge, there has been no prior work attempting to study the influence of these factors on soft error vulnerability of multicores.

The goal of this work is to fill a critical void in the characterization of soft error vulnerability of multicores running multi-threaded applications. We use AVF to enable our study. Recalling from chapter 3 that AVF is directly proportional to the total error rate of the system, it serves as a good metric that is reasonably easy to capture and also is a good indicator on the overall error rate. To enable our AVF analysis, we develop a very detailed framework on an execution-driven full system (including OS activities) simulator. Using this infrastructure we study the performance and vulnerability of multicore platforms with different number of out-of-order cores (up to 8), having an on-chip 2-level cache hierarchy, running Solaris 10 using ten multi-threaded applications from different benchmark suites. We make the following observations and contributions from our experiments:

- Considerable variation in soft error vulnerability exists across the cores when running multi-threaded applications. In some executions we found the absolute difference in AVFs between cores to be as high as 50%. This variation in AVF suggests the need for differential hardening/protection across the cores so as to reduce the overheads (performance and power) on every core.

- Separation of the time and vulnerability between user and kernel mode shows that the latter has a significant portion in the overall soft error vulnerability which cannot be ignored. Our inspection into the kernel code suggests that

synchronization between threads (due to locks and barriers) has a significant impact on the overall vulnerability.

- We also find interesting results when we vary the *multicore parameters* - the number of cores and the number of threads running on these cores. When we increase the number of cores linearly with the number of threads, while performance does improve we find in many cases the vulnerability grows even faster than the speedup. In several cases we find that running the same number of threads on fewer cores (e.g. 8 threads on 4 cores instead of 8 cores) actually gives a much lower vulnerability while the speedup is not affected as much.

- While one could exploit the above observation to statically pick a choice of multicore parameters that gives the best performance-reliability tradeoffs, we notice that there exists dynamic variation in the overall vulnerability of the system (periods of high and low vulnerability) during an application run, which can be exploited as well. We develop runtime techniques that works together with the OS scheduler to tune the number of cores on which the application threads run. Compared to the configuration offering maximum throughput, these techniques decrease the overall soft error vulnerability by as much as 37% with less than 5% loss in performance.

Section 6.4 gives the simulation infrastructure, the workloads and the issues associated with doing AVF analysis for multicores. Section 4.3 shows the performance and AVF across the threads of an application running on a 8-core system. The results from varying the multicore parameters are given in Section 4.4. Section 4.5 presents our runtime adaptive technique to adapt parallelism for reliability and performance. Discussion of related work is presented in Section 4.6, and we conclude with a summary of results in section 6.7.

(A) System Parameters. Latency in Parenthesis

Parameter	Value
Pipeline Width	2
Fetch Queue/ROB/LSQ/ISQ Size	96/96/64/32
Int ALUs/Mult/Div	2 (1,4,20)
FP ALUs/Mult/Div/Sqrt	2 (2,4,12,24)
L1 D/I-Cache	32KB, 2-way, MSI
L2 Unified	2 MB, 4-way, MOSI
Network	Point-to-Point, 13 cycle latency
Memory Latency	200 cycles

(B) Workloads

Parameter	Input
Ammp/Applu Equake/FMA3D/Galgel	SpecOMP
Barnes	16384 particles (Splash-2)
Cholesky	tk29.0 (Splash-2)
LU	2048 × 2048 (Splash-2) 16 × 16 blocks
Ocean	514 × 514 grid (Splash-2)
Specjbb2005	Java-based benchmark

Table 4.1. (A) Per-core and multicore platform parameters. (B) Benchmarks used in our analysis

4.2 Experimental Methodology

4.2.1 Simulation Infrastructure

All our experiments were run on the GEMS full system simulator [71] which tracks both application and kernel activity and their interaction in detail. The Ruby module in GEMS helps support the cache coherent CMP system while Opal models the out-of-order (OOO) processor used in our evaluation. While most parts of the system remain similar to the UltraSparc IIICu CPU (SPARC V9 ISA running Solaris 10), Opal has been modified to support a more generic pipeline discussed below. Table 6.1A gives the overall system setup. The workloads, see Table 6.1B, includes a set of scientific benchmarks (from the SPLASH-2 and SPECOMP 2001 suites) and SpecJbb 2005, a representative commercial application. The scientific benchmarks were run for 100M user instructions and SpecJbb was run to complete 400 transactions. All the benchmark runs were done after warming up the caches. Each run takes several hours to days to complete due to the complexity associated with vulnerability tracking across cores, discussed in detail below. To account for the variations across simulation runs when employing multi-threaded workloads, we ran each simulation several times and report the average value across these runs.

4.2.2 System Configuration

Our system is configured similar to existing homogeneous multicore architectures [54, 58, 50]. We perform our analysis on 2, 4 and 8 core CMPs. Each core incorporates a P6-style pipeline shown in Figure 4.1 [97]. Instructions get fetched and decoded in-order in the front-end of the pipeline. Once decoded, the instructions get dispatched simultaneously to the Reorder Buffer (ROB) and issue queue (ISQ). Load and store instructions get dispatched to the load-store queue (LSQ). ISQ handles the out-of-order issue of instructions for execution. ROB maintains the instruction states as these instructions pass through different pipeline stages. This is required for handling situations like branch mis-predictions when instructions get flushed from the pipeline depending on their state. Once instructions complete execution, their results get written in the ROB. Commit occurs in-order, after this, when the instructions become the oldest in the pipeline. At this point, their results get written to the architected register file (ARF). Till then, the ROB maintains the result in its entries. The L1 caches are private while L2 is shared across all the cores.

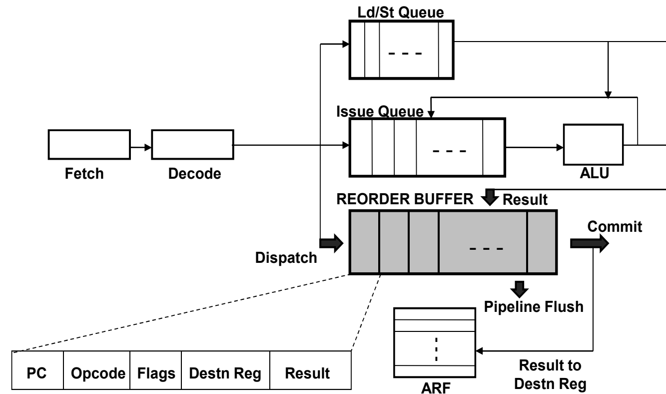


Figure 4.1. Pipeline design within each core. Figure highlights pipeline events specific to the Reorder Buffer (ROB) besides showing sub-components within a ROB entry.

4.2.3 Multicore AVF Analysis Framework

We have earlier introduced AVF, in chapter 3. To recall, AVF gives the fraction of the total bits that are vulnerable to soft errors. It captures the error masking

that exists at the architectural and micro-architectural level in the system.

While chapter 3 looked at providing a runtime AVF tracking infrastructure, here we look at measuring AVF over a certain interval of application execution. Hence we track of events like register and memory overwrites that lead to unACE-ness and thereby lower AVF. Tracking events like register overwrites, where the writes can be temporally separated by hundreds to thousands of cycles, require the need to keep track of the instruction state even after they leave from the pipeline to compute the AVF. Hence we maintain a *Post Commit Buffer (PCB)*, similar to the analysis window proposed in [75], in which the state of retiring instructions are maintained. In our simulations, the PCB in each core was sized at 8192 entries, to tradeoff between accuracy of tracking and simulation speed. Any instruction that is not classified as unACE before its entry in PCB is reused, is an ACE instruction and it contributes to the AVF.

Merely tracking the per-instruction characteristics is not sufficient to compute AVF accurately. This is because instructions incorporate sub-components of varying importance. For example, the opcode (which identifies an instruction) of both ACE and unACE instructions needs to be correct while the requirement for their results to be correct is there for ACE instructions only. Consequently the ACE and unACE classification is extended to the bit-level granularity. Instruction bits whose correctness affect the final output are ACE bits, while bits whose state does not affect the final correctness are unACE bits [75]. Note that unACE instructions could have ACE bits (opcode bits) as part of their entry. In addition, as Figure 4.1 shows, different sub-components in an entry within a structure can have varying sizes and these sub-components can get filled at varying times over execution. In the figure, the Program counter (PC), opcode, flag bits and destination register identifiers of the ROB entry get filled at dispatch while the result is available only when the instruction completes execution. This implies that for AVF computation to be accurate, per-bit tracking of the residency times of bits in the different structures is essential. Note that ACE bits residing for longer periods increase the AVF of the structures. Combining these factors, the AVF of a structure becomes the fraction (given as a percentage value) of the total time ACE bits reside in the structure, given as

$$AVF = \frac{ACE\ Bitcycles\ [= \sum(ACE\ Bit * Occupancy\ Time)]}{Total\ Bitcycles\ [= Total\ Bits * Total\ Time]} \tag{4.1}$$

The CMP environment coupled with the multi-threaded nature of the applications greatly increase the complexity of tracking ACE and unACE bits. Memory overwrites occur at multiple levels of cache hierarchy (L1 and L2) and interval between these events can be long as well (data was overwritten 10K to even 100K cycles later) which needs to be tracked and accounted at a specific core for the AVF computation to be accurate. Further, coherency state transitions in the caches, due to the multi-threaded nature of the workloads, need to be monitored as well since certain transitions (like Modified to Invalid transitions) lead to data values becoming invalid immediately after being written (with no intermediate reads) and bits corresponding to the store instructions that did the write operations in the cache are unACE. In our work, we track all these factors that affect AVF in a multicore platform. Further, we also accounted for the unACEness resulting from instructions that occur only on the back-edge of these unACE writes [14].

For a 8-core system running 8 application threads across all cores, we found that the number of vulnerable ACE bits in the ISQ and LSQ, large pipeline components, to be only about 38% and 10% of what was found in the ROB, across all workloads. Also instructions spend significant percentage of their lifetime in the ROB [97]. The ROB thus has a large impact on the overall soft error vulnerability and it captures the major trends in our analysis. Therefore we focus on the AVF analysis of the ROB as representative of the processor in the rest of this chapter.

4.3 Multicore ROB AVF Analysis

In this section we look at the AVF variations across a 8 core CMP running 8 application threads. We identify the factors that contribute to the significant variation in AVF across the cores including those dependent on the nature of instructions that run in each of the cores and those associated with the multi-threaded nature of the applications. Since AVF affects the overall error rate of the cores, refer section 4.1, protection mechanisms that are unaware of these variations

end up providing inefficient (consuming more power as they provide protection for cores with low AVF) or insufficient levels of protection (not meeting the reliability requirements for cores with high AVF). The section further highlights that kernel AVF is a significant contributor to the overall AVF of multicore platforms when running multi-threaded applications and users need to be aware of their impact to prevent compromising the reliability of the overall system.

4.3.1 AVF Breakdown for the 8-core CMP

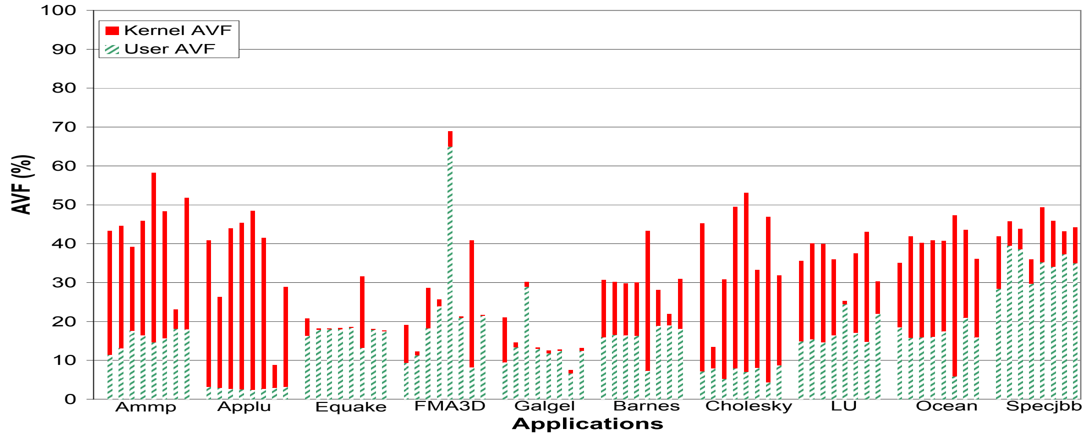


Figure 4.2. AVF variations across different cores of 8-core CMP. The height of the bar corresponds to the overall AVF of a core.

Figure 4.2 shows the per-core AVF for the 8 core platform running different applications. The total AVF is split into user and kernel AVF to highlight their individual contribution. We observe that kernel code is a major contributor to the overall AVF, in some cases their portion being more that 90% (`applu`). Further the individual portions of the user and kernel AVF vary widely across the cores as well (as seen between cores 5 and 7 in fma3d). When deciding the necessity for soft error protection techniques (to meet reliability requirements) versus the performance and hardware costs involved in incorporating them, this result shows that the effects of kernel code cannot be ignored in such an analysis.

Figure 4.2 also shows that significant difference in AVF exists among the cores, when running each of the applications, and this could be as high as 40%-50% as seen for `applu` and `fma3d`. Given the significant variation in AVF across the cores, we evaluate the Mean Instructions to Failure (MITF) [122] for the cores. MITF

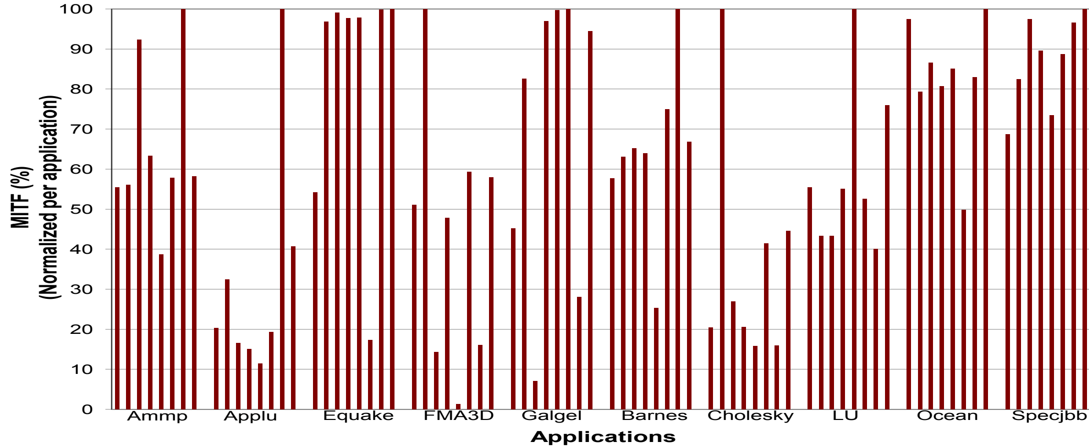


Figure 4.3. MITF variations across different cores of 8-core CMP. The significant variation in MITF across cores imply a need for protection techniques to be aware of these variations to provide better performance-reliability tradeoffs.

is a metric used to capture the amount of useful work done between failures and hence a higher value is preferred. As pointed out in [122], for a system involving homogeneous cores running at the same clock frequency, $MITF \propto (IPC/AVF)$ where IPC is the number of instructions committed per cycle. For multi-threaded applications, useful work manifests as the number of user (application) instructions committed [126] , hence $IPC = IPC_{user\ code}$ while AVF corresponds to the overall AVF of the core. Given each core commits varying number of instructions, large variations in MITF across cores enable users to determine the cores that are more important to protect. Figure 4.3 shows the normalized MITF (higher the better) across the different cores and portrays that the difference in MITF between cores can be greater that 90% for certain applications (`fma3d`, `galgel`). This observation motivates the need for runtime AVF monitoring to selectively protect the cores for improving their MITF. Such solutions reduce the performance and power overheads of protection mechanisms that are applied uniformly across cores.

4.3.2 Factors affecting AVF variation

Next, we analyze different application characteristics that contribute to the per-core AVF and also investigate causes for their variation across the cores. While factors like branch misprediction, L1 and L2 capacity misses affect the AVF of

individual cores, the effect of these parameters has been studied [75, 122]. We restrict our analysis to factors that characterize multi-threaded applications.

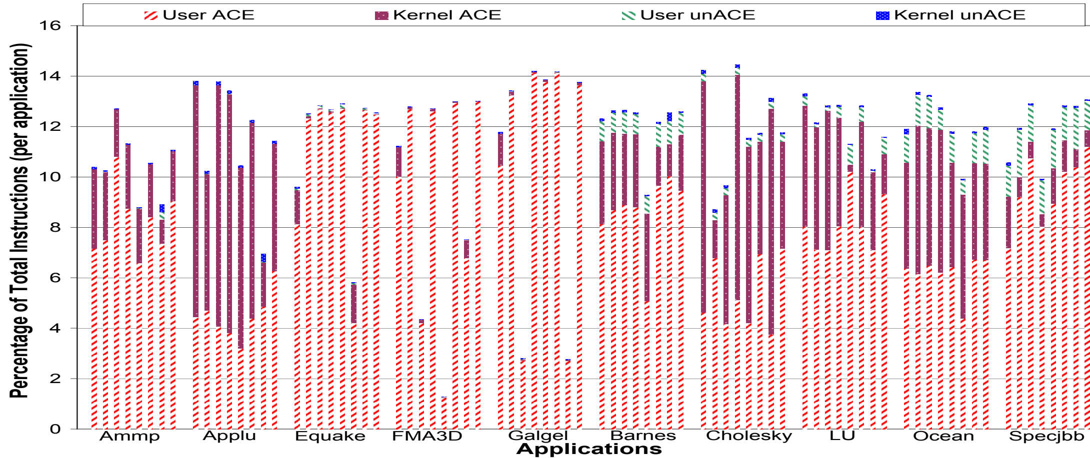


Figure 4.4. ACE/unACE instruction classification. The height of each bar corresponds to the fraction of the total instructions that retire from a core.

4.3.2.1 Instruction Characteristics

We first look at classifying the user and kernel instructions flowing through each of the cores into ACE and unACE (excluding nop instructions which are unACE), as shown in Figure 4.4. The unACE committed instructions arise from three factors - namely instructions whose results gets overwritten in architected registers and store instructions which get overwritten or invalidated in memory. For example, `specjbb` has a lot of store overwrites (10% of total stores) in all its threads contributing to its unACEness.

Looking at Figures 4.2 and 4.4, it becomes apparent that on average the per-instruction contribution to overall AVF differs between the kernel and user code. For example, in `cholesky` this value for the kernel is 4 times the user code's value. One reason for this difference is the kernel code having two times more instructions in the shadow of L1 misses than the user code. Instructions in the pipeline (have ROB entries) from the time a cache miss occurs till it is handled are said to be in the *shadow of that miss*. These instructions stay in the ROB for longer periods and have higher occupancy times compared to the other instructions due to the time involved in handling the miss and increase the AVF of the structure. The figures also provide insights for designers looking at variable protection levels across cores.

For example, in `galgel` core 3 executes fewer instructions than core 4 but it has a higher AVF. Consequently, runtime mechanisms to enable and disable protection based on monitoring simple events like the total user instructions committed will not be sufficient. Such a scheme, in our example, would end up providing more protection for core 4 which is not optimal since core 3 has a lesser MITF and is more likely to fail earlier (Figure 4.3).

An important class of instructions seen in multi-threaded applications are atomic instructions (`specjbb` executes 160,000 atomic instructions during execution) required for achieving synchronization in hardware. Different architectures handle them differently once they enter the pipeline which significantly impacts AVF of the cores. In the UltraSparc IIICu (SPARC V9 ISA) architecture we study, `ldstub, casa and swap` are atomic instructions which serialize execution [123], and the pipeline is stalled until these instructions complete execution. Later instructions that get fetched are flushed out of the pipeline. This is not the case with x86-based systems. While not the main theme of this chapter, designers need to be aware of these factors when investigating the soft error vulnerability of multi-threaded applications.

4.3.2.2 Impact of coherency misses

An important characteristic of multi-threaded workloads are coherency misses which occur due to the read-write and write-write data sharing that exists between the threads [34]. The data sharing leads to cache lines getting invalidated (in the L1 caches) and hence later accesses to these cache lines experience a miss. Unlike capacity or conflict misses which can be reduced by increasing the cache size and associativity, coherency misses offer an interesting dimension to the analysis since they are determined by the number of threads contending for shared data in the applications and the nature of sharing that exists between threads. Figure 4.5 shows the impact on ROB AVF due to user and kernel instructions in the shadow of L1 coherency misses. A significant portion of the overall AVF (74% in `equake`) are contributed by instructions in the shadow of coherency misses. In some applications (e.g., `specjbb`), while the fraction of the total instructions under L1 coherency misses are only moderate (14% in `specjbb`), their contribution to overall AVF is significantly higher (as seen in Figure 4.5). These instructions occupy

their ROB entries for significantly more cycles than otherwise increasing the AVF. With the increasing number of cores on-chip and drive towards multi-threading, there is a definite need to reduce the latency associated with the coherency misses which would reduce the soft error vulnerability of the cores as well. Also techniques like pipeline flushing [130] avoid instructions from idling in the ROB when cache misses occur. These solutions can be adopted to reduce AVF as well.

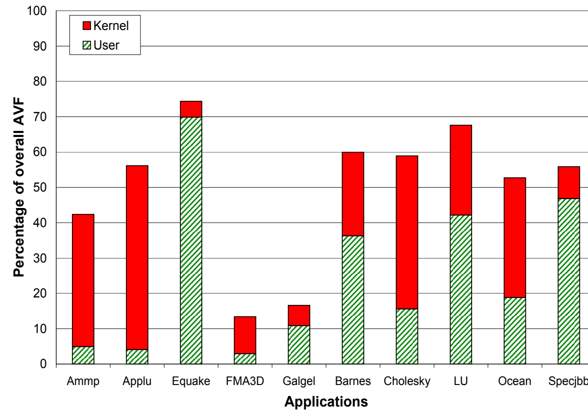


Figure 4.5. Fraction of the overall AVF due to instructions (user and kernel) in shadow of L1 coherency miss. The values are averaged across the 8 cores.

4.3.2.3 Impact of spinlocks

Thread synchronization is an important factor that limits the scalability of applications (in terms of reducing parallelism). Synchronization between threads is done using locks and barriers. Locks are used to provide exclusivity inside critical sections of application code while barriers ensure that all the threads reach a certain point in the application code before they proceed further. If a thread waits for a lock (held by another thread) to be released or reaches a barrier before other threads do, the kernel can schedule another thread or it *spins* (busy-waits) on the core. Spinning, or the spinlock routine, involves the kernel executing an idle loop that repeatedly checks if a condition is true (lock is released or when all threads have reached the barrier). Higher the lock contention between threads in an application or the more time a thread waits at a barrier, lower is the useful work done with respect to time. Figure 4.6 shows the fraction of kernel AVF contributed by spinlocks due to synchronization. We use the algorithm mentioned in [124] for spinlock detection. The figure shows that spinlocks contribute a very

significant part of the kernel AVF for most of the applications. We observe that in applications like `ocean` which incorporates a lot of barriers in its code, the spinlock routine is invoked by all cores. Other studies [27] have proposed schemes to predict the criticality of threads and using it to slow (frequency-scale the other cores) the other threads accordingly. However slowing down cores can increase the ACE instruction occupancy time in the ROB which increases its AVF. Its been shown that voltage scaling can be non-ideal when reliability impact is considered [103].

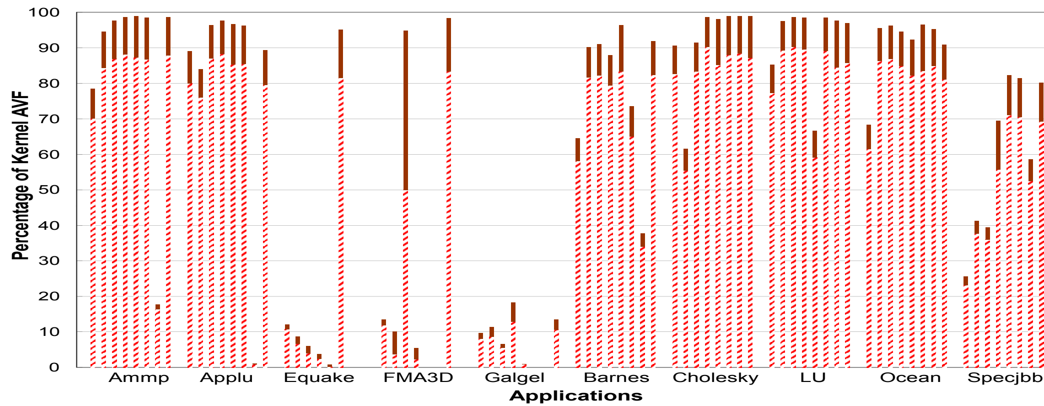


Figure 4.6. Fraction of kernel AVF due to spinlocks. The solid portion in each bar gives the unACE portion of the "test and test-and-set" code.

An efficient implementation of kernel spinlocks usually involves a "test and test-and-set" piece of code (as in the Linux and Solaris kernels [28]) where the test portion loops multiple times until the associated flag is reset. At this point, the test-and-set portion of code is entered, in which atomic instructions (`ldstub, casa or swap`) refetch and test the flag. Only when this check is successful, does the kernel come out of the spinlock routine and allows the thread to proceed. Such a design reduces the performance overhead associated with continuously executing the "test-and-set" code since the atomic instructions cause cache line invalidations when reading the flag bit leading to cache misses which affects performance. Notice that the test portion gets executed multiple times while the test-and-set executes much less frequently. Since the "test-and-set" rechecks the work done by the "test" portion, instructions which are part of the "test" portion can be classified as unACE since an error in their result does not affect final output. Figure 4.6 shows the effect of this classification on the spinlock AVF. Recall from section 4.2.3

that unACE instructions incorporate ACE bits (e.g., opcode bits) as part of their entry. These bits prevent the spinlock AVF from decreasing significantly and as a consequence the effective spinlock AVF remains high.

Application users also look at *pinning* (binding) threads to specific cores to reduce cache pollution (cache locality gets affected when different threads run on a core) or to handle interrupts more efficiently [55]. We investigated the effect of thread pinning to cores (one thread to one core) on the spinlock AVF. We observed that such pinning further increased the ROB AVF (as high as 34% more for `LU` between non-pinning and pinning of threads). As threads are pinned to specific cores, the kernel scheduler avoids assigning a different thread to run on cores in which the running thread is blocked (due to locks or barriers). Instead it invokes the spinlock routine. Thus the reliability tradeoffs due to pinning mandates serious consideration when weighed against any performance benefits it offers. More detailed results are omitted for brevity.

Multicore Configurations

Configuration	**Label**
8 threads on 8 cores	8T_8C
4 threads on 4 cores	4T_4C
8 threads on 4 cores	8T_4C
2 threads on 2 cores	2T_2C
8 threads on 2 cores	8T_2C

Table 4.2. Different configurations used in our evaluation.

4.4 Impact of Multicore parameters

In this section, we look at the impact on performance and reliability when running multi-threaded applications under differing settings of the multicore parameters. The configurations investigated are given in Table 4.2. Till now our study was restricted to the first configuration. The rest correspond to typical choices made for running multi-threaded applications on multicore platforms. The following analysis throws light on the fact that while multicore platforms have increased the choice of configurations, there are reliability compromises associated with these choices. *Even for a given platform, the choices made dramatically impacts the soft error*

vulnerability of the underlying hardware. Where a designer would have required protection mechanisms to meet reliability demands, a change in configuration might provide it at lower power, performance and hardware overheads.

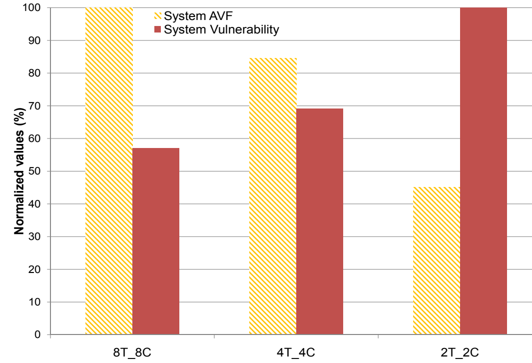


Figure 4.7. System AVF and System Vulnerability variations across three different multicore configurations when running fma3d. The figure shows that while the System AVF is highest for a configuration(8T_8C) it does not manifest in a higher System Vulnerability, which is highest for 2T_2C.

4.4.1 System AVF and System Vulnerability

While AVF is directly proportional to the soft error rate ($FIT_{effective}$), see section 4.1, it is normalized with respect to time. It gives an average per cycle vulnerability of a structure to soft errors. Therefore when analyzing the reliability impact of two different configurations with varying runtimes, studying the impact on AVF, alone, does not provide the full picture. To emphasize the point, consider a structure whose size is 1000 bits. A configuration that leads to 10 ACE bits occupying the structure every cycle for the entire runtime of 100 cycles has 1000 ACE bits ($10*100$) in total and hence its AVF is 0.01 ($(10*100)/(1000*100)$) while a different configuration that leads to 50 ACE bits occupying the structure for the runtime of 10 cycles has 500 ACE bits ($50*10$) in total and AVF is 0.05 ($(50*10)/(1000*10)$). Note that while the AVF is higher in the second case, the total ACE bits (vulnerable to soft errors) is actually lesser in that configuration. Hence we also capture the impact on the total ACE BitCycles (refer equation 4.1), referred as *cumulative vulnerability*, to better quantify if a configuration leads to a lower vulnerability to soft errors. For reliable operation, along with a lower cumulative vulnerability, it is

desirable to have a low AVF as well since designers use AVF to establish the error rate of the system and the susceptibility of the underlying system to soft errors is high during periods when AVF is high.

Depending on the configuration, Table 4.2, the number of cores on which the application runs differ. Hence we characterize the total AVF (sum) across all cores, referred as System AVF, and the cumulative vulnerability across all cores, referred as System Vulnerability. Note that while AVF is a fractional value, System AVF is not. System AVF is an indicator of the per-cycle vulnerability to soft errors of a multicore platform while System Vulnerability gives the cumulative ACE BitCycles across all cores of the muticore platform an application runs on.

To illustrate the need for the following analysis, we use Figure 4.7 which shows the System AVF and Vulnerability of 3 different configurations when running `fma3d`. It clearly emphasizes the point that while the 8-core configuration has a higher System AVF, the 2-core configuration runs for a much longer period and ends up with a higher System Vulnerability. It becomes apparent that the specific configuration chosen can dramatically vary the reliability characteristics of the underlying platform (55% variation in System AVF and 40% in System Vulnerability across the configurations as seen in the figure).

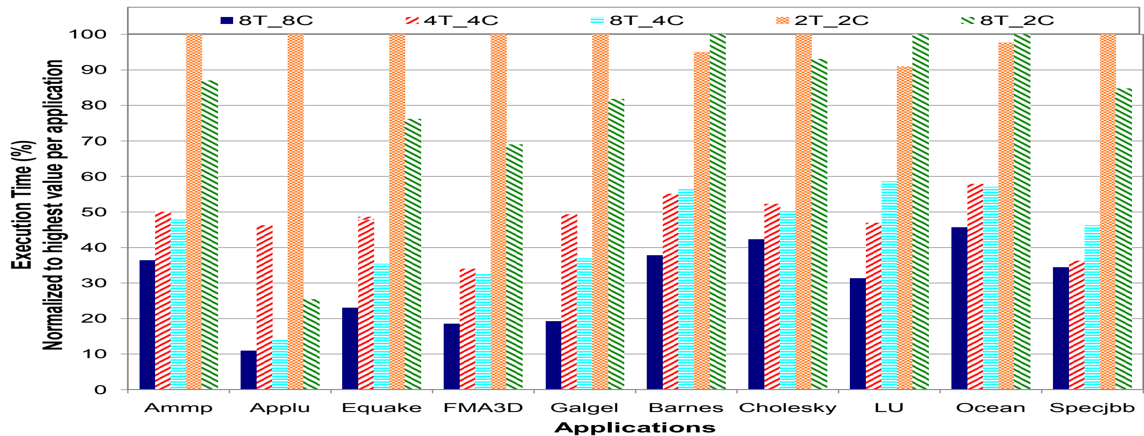


Figure 4.8. Execution time variation across the different multicore configurations.

4.4.2 Performance Impact

We first investigate the performance impact due to running the applications in the different configurations. Figure 4.8 indicates that when performance demands are required to be met, running the application on more cores is helpful. The throughput advantages offered by multicores which allow applications to exploit their TLP becomes an significant factor in reducing the overall runtime. In applications like `equake`, the aggregate overall system IPC decreases by a factor greater than 50% between the 8T_8C and 4T_4C configurations which continues to exist between the 4T_4C and 2T_2C configuration as well. For the under-provisioned configurations (8T_4C and 8T_2C), the number of context switches between threads have a big impact on increasing the execution times (in `ammp`, the number of context switches between threads in the 8T_8C configuration is 50% lesser compared to the 8T_4C and 80% lesser compared to 8T_2C configurations).

4.4.3 Impact on System AVF

Figure 4.9 shows the varying impact on system AVF across configurations. It is apparent that greater availability of cores increases the overall System AVF. In certain cases like `applu`, `barnes` and `cholesky`, System AVF increases by a factor larger than the scaling with respect to cores. The kernel AVF (due to spinlocks) is a major factor for higher System AVF. As the number of cores decrease, it reduces the period for which OS needs to spin due to the availability of application threads to run in the cores. On average across all applications, the spinlock vulnerability decreases by a 70% and 81% when the cores available decrease to 4 or 2 for running 8 application threads. The impact of the coherency misses decreased as well with lesser number of instructions in the shadow of coherency misses. On average, there was a 50% reduction the AVF contribution from instructions in the shadow of coherency misses between the 8C_8T and 4C_4T configurations. A key observation is the higher user System AVF whenever more cores are available. Therefore, even if alternate solutions are developed to avoid the use of spinlocks, System AVF would still be high when applications are run on more cores. The main reason for this being the mismatch between performance scalability of applications and available hardware. For applications like `cholesky`,`ocean` and `specjbb`, running

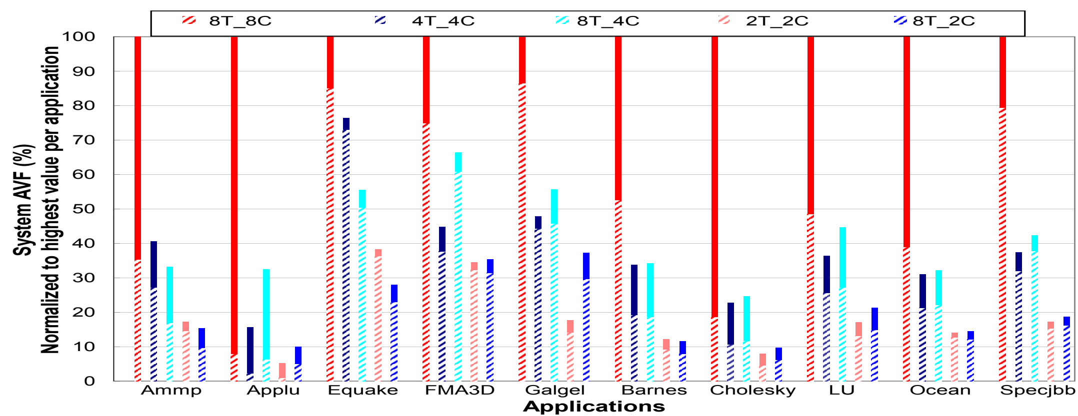


Figure 4.9. System AVF across the different multicore configurations. Solid portion gives the kernel code contribution while striped portion is the user code contribution.

on 4 cores provides almost 80% of the 8-core performance. The lack of throughput increase the instruction occupancy time in the ROB resulting in the rise in AVF.

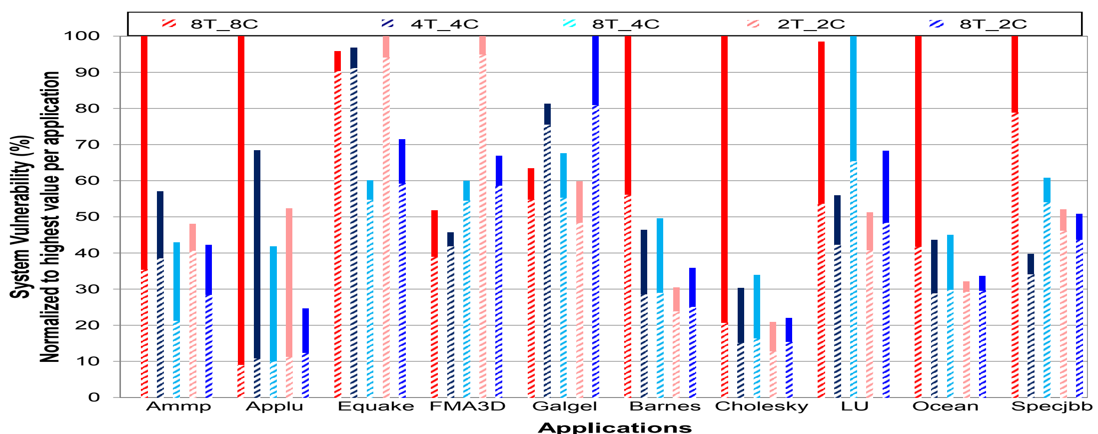


Figure 4.10. Vulnerability across the different multicore configurations. Solid portion gives the kernel code contribution while striped portion is the user code contribution.

4.4.4 Impact on System Vulnerability

Figure 4.10 which shows the System Vulnerability clearly points out that decreasing the available cores doesn't imply a reduction in the System Vulnerability. While the 8 core run had the highest AVF, in applications like `equake`, `fma3d`, `galgel` and `LU`, the 4 or 2 core configurations lead to higher System Vulnerability. Factors like

increased number of cache misses, lower ILP compared to the TLP combined with the increase in runtime, as seen in Figure 4.8, increase the System Vulnerability. This emphasizes that the configuration choices with respect to optimizing the reliability parameters (AVF and vulnerability) aren't straight-forward.

Figures 4.8, 4.9 and 4.10 show that for a specific number of cores, the number of application threads do not seem to categorically impact one parameter (for example, running 4 or 8 application threads on 4 core configuration). In cases like `applu` and `galgel`, enough TLP exist in the applications for it be the more optimal design choice while for `LU` and `specjbb`, ILP works out better for performance and System Vulnerability.

Core-allocation parameters

Parameter	**Value**
CB_Interval	10000 cycles
Spin_Thresh	10% of period
ROB_Thresh	48 entries
IPC_Thresh	0.5

Table 4.3. Different parameters associated with core-allocation schemes

4.5 Core-Allocation schemes to optimize Performance Reliability tradeoffs

The last section showed that choosing a specific configuration comes with associated performance-reliability tradeoffs which might be sub-optimal depending on the requirements. Further, it revealed that decreasing the number of cores does not guarantee a reduction in the System Vulnerability. Given the significant dichotomy in performance and reliability that exists across these configurations, careful selection is required to identify the specific design that meets both requirements. This section proposes simple runtime techniques (allocation policies) that vary the number of cores on which the application threads can run based on monitoring core-specific events. The need for such techniques is motivated by the dynamism that exists in throughput and System Vulnerability during an application run. Figure 4.11 tracks the runtime variation in these metrics (throughput given by the

aggregate $IPC_{usercode}$) when running `LU` in the 8T_8C and 8T_4C configurations. It is apparent that there are phases during execution in which each of the configurations provides significant benefits if applications choose to run in them and choosing one over another statically is not optimal. The techniques proposed have low overhead, only requirement being the spin-detection logic [124] and counters for tracking per-core IPC and ROB utilization (average number of instructions occupying ROB entries per cycle).

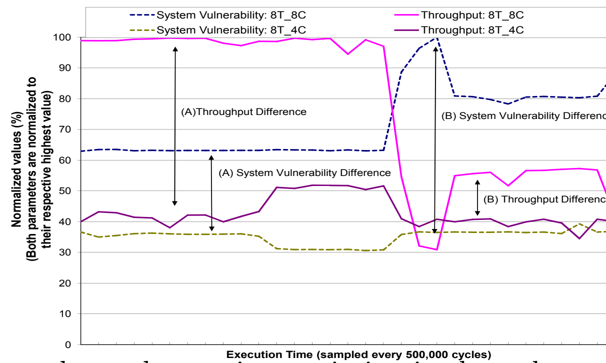


Figure 4.11. Figure shows the runtime variation in throughput (aggregate IPC across cores) and System Vulnerability when executing `LU` in 2 configurations. It highlights phases that can be exploited for decreasing System Vulnerability without losing performance. In (A)Throughput Difference > System Vulnerability Difference. Hence 8T_8C is more optimal in this phase. In (B)System Vulnerability Difference > Throughput Difference. Hence 8T_4C is preferred in this phase.

Varying the number of application threads itself at runtime (for example, varying the number of threads in `ammp` from 4 to 2 or 8 during execution) involves application code redesigning and hence is not considered here. The techniques proposed in this section operate on a given number of application threads (in our evaluation, we assume 8 application threads). They work with the OS scheduler to vary the number of cores on which the threads run such that performance and reliability losses are minimized. Periodically (every CB_Interval cycles) the counters are reset to track application behavior over time. The decision to vary the number of cores is taken at context-switch boundaries based on the application behavior in the last period. To show the efficiency of the schemes, we highlight the impact of the schemes on different parameters against the 8-thread configurations from table 4.2. All the schemes assume a minimum of 2 cores on which the application threads can run. Values corresponding to different parameters (mentioned below)

used by the schemes are given in table 4.3.

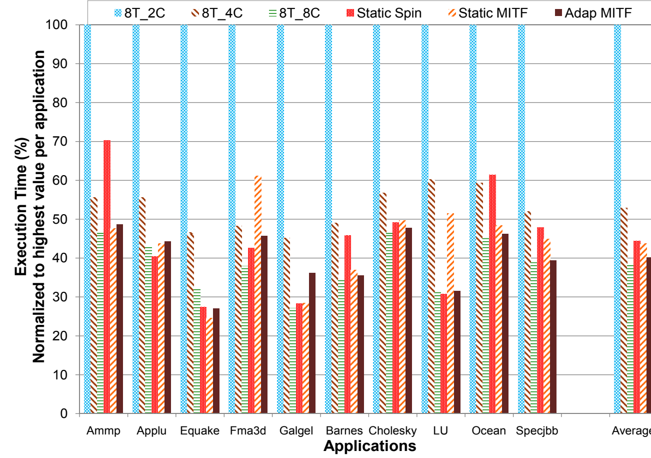


Figure 4.12. Execution time comparison across configurations revealing the low overhead of the core allocation schemes.

4.5.1 Static Spinlock

Given that spinlocks become a predominant factor with the increase in the number of cores, in this scheme we detect periods when the OS starts to run the spinlock routine on a core. Whenever a core is detected to spin for periods longer than a pre-defined threshold (Spin_Thresh), using the spin-detection logic, the scheduler would bind the application threads to a lesser number of cores. This allocation of cores is maintained for a certain period (CB_Interval) after which the scheduler may reassign applications threads to the core.

4.5.2 Static MITF

It is desirable for cores to have a high MITF value. Given that $MITF \propto (IPC/AVF)$ [122], whenever IPC drops below IPC_Thresh (pipeline width is 2 in our system) and AVF is high, threads are allocated to a lesser number of cores. To enable runtime tracking of periods when AVF is high, we monitor the average ROB utilization in each core, which directly corresponds to the AVF of the structure [118]. The ROB utilization threshold is set to ROB_Thresh (ROB size is 96 entries) This scheme operates on top of the Static Spinlock scheme.

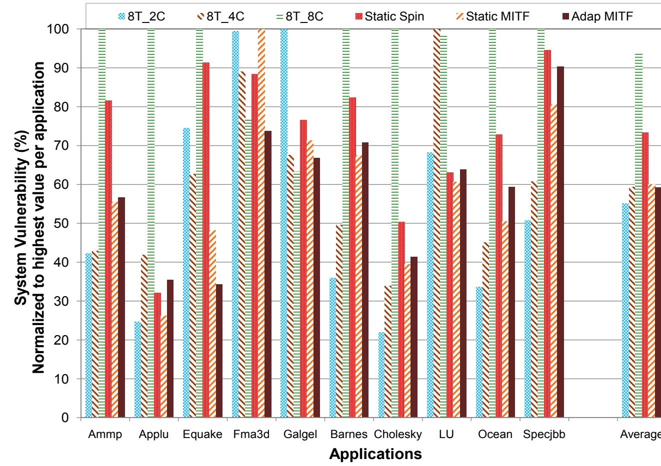


Figure 4.13. System Vulnerability comparison across configurations. Compared to 8T_8C the core allocation schemes reduce System Vulnerability by greater than 35% (MITF schemes).

4.5.3 Adaptive MITF

Fixed threshold parameters used by the Static MITF scheme might not be optimal in terms of both performance and reliability across all the applications. Hence we look at changing the threshold IPC and ROB utilization parameters according to application behavior. The IPC_Thresh is set to the average IPC across the cores on which the application runs while ROB_Thresh is set to average ROB utilization across the cores. The spin-detection threshold still remains static since making it adaptive didn't provide any significant gains. For the adaptive scheme, the IPC and utilization values are transferred periodically (CB_Interval cycles) between the cores with minimal impact on performance.

Figure 4.12 shows the performance impact of choosing the different core-allocation schemes. Compared to 8T_8C configuration, which has the lowest execution time, the performance overhead varies between 13% for the static spinlock scheme to 4.7% for the Adaptive MITF scheme. Looking at Figure 4.13, which gives the impact on System Vulnerability, compared to the 8T_2C configuration (which has the lowest System Vulnerability) the static spinlock scheme increases vulnerability by 20% while the adaptive MITF does it by 6%. In both cases (performance and System Vulnerability), the comparison is done against configurations which were more optimal for those parameters. To show the efficiency of the core allocation schemes, note that the Adaptive MITF decreases execution time by 60% com-

pared to the 8T_2C configuration and System Vulnerability by 37% compared to the 8T_8C configuration.

Figure 4.14, which gives the System AVF, shows that while the core-allocation schemes have a higher System AVF that the 2-core configuration, they bring it down by as much as 45% (adaptive MITF, growing up to 49% for static MITF) compared to the 8-core configuration. The effectiveness of the schemes stem from being able to detect periods when the throughput improvement offered by additional cores are small while the soft error vulnerability remains high in those periods. In certain cases, like `fma3d` and `LU`, the static MITF scheme fails to capture the application behavior. The Adaptive MITF scheme, in general, does better at tracking application characteristics.

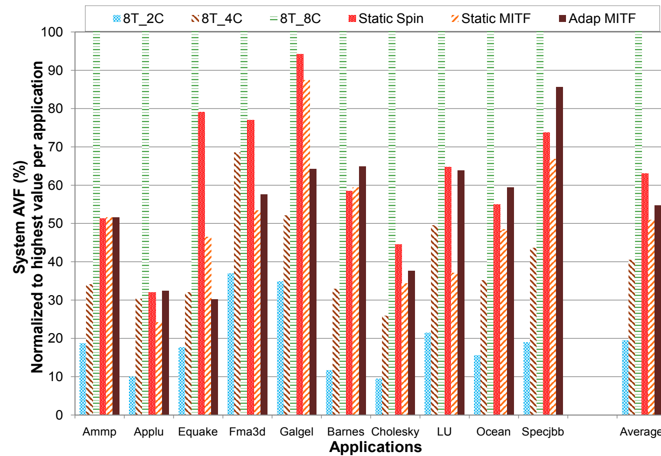


Figure 4.14. System AVF comparison across configurations. Given that AVF cannot be significantly increased, these graphs show the core allocation schemes to have only a moderate impact on the System AVF compared to the 8T_8C configuration.

4.6 Related Work

Several techniques have been proposed in literature to tackle the impact of soft errors in single cores. Broadly these solutions can be classified as works that operate at the circuit and architectural level [30, 122]. The architectural solutions to detect and recover from the soft errors operate at different granularities. These include techniques that replicate instructions, do redundant execution of threads or look at cycle-level lock-stepping of processors to verify execution of the same execution

stream. While most of these techniques have a moderate performance overhead, like Simultaneous Redundant Threading (SRT) [91], the techniques tended to focus mainly on single-threaded applications.

Solutions for tackling soft errors in the CMP domain include works like [46, 87, 102]. Some of the initial works like [46] looked only at single-threaded applications and the scalability of these solutions, as the number of cores increase, is limited. While lock-stepping of cores provided a simple solution to overcome the non-determinism associated with memory access, due to multiple executions being done in parallel, it is costly to incorporate. The technique proposed in [131] looks at reducing the soft error vulnerability in a CMP environment when running multi-programmed workloads using a utility-driven function that throttles and allocates pipeline resources dynamically based on the cache partitioning between the workloads. [43] compares the MITF of thread-level redundancy and instruction-level redundancy when running SPEC2K workloads for a 2-core cluster-based CMP. Works like Reunion [102] looked at more cost-effective solutions using master-slave core pairs to enable greater flexibility for checking values. Fingerprint messages are exchanged across the cores to check the correctness of data and commit values to architected state. While the performance overhead is around 9%, the technique requires checker logic and hardware for computing and communicating the fingerprint across cores. Mixed-mode reliability, proposed in [125], works on top of a Reunion-enabled system to allow applications with varying performance and reliability requirements to co-exist. Thread Level Redundancy (TLR) as proposed in [87] is another technique that is similar in motive to Reunion, though TLR looks beyond soft errors as well. To allow the threads to run independently, they require a Post Commit Buffer (different in purpose from the one mentioned here) that is about one-third the size of the L1 cache besides having bandwidth and rollback issues as well. Our study is complimentary to these techniques since choosing a more ideal configuration, with respect to soft error vulnerability, can help enhance the performance of system as it can be done based on the reliability level required.

4.7 Summary

This chapter presented the first study on the impact of soft error vulnerability of a multicore platform running multi-threaded applications. The detailed analysis capturing the AVF variations across different cores highlighted the need for developing reliability solutions with differing levels of protection across cores. Kernel code was shown to be a significant contributor to the overall vulnerability of the system whose reliability impact cannot be ignored when running multi-threaded workloads. Varying the multicore parameters to capture the System AVF and System Vulnerability illustrated that the associated performance-reliability tradeoffs are not straight-forward. Across different applications, we observed that certain configurations led to a dramatic increase in the System AVF not matched by the speedup they offer while other configurations with lesser number of cores had a sub-optimal impact on the System Vulnerability. To meet the dual requirements of high throughput and low vulnerability not readily met by the statically chosen configurations, we develop runtime techniques that identify the ideal execution phases for optimizing performance and vulnerability. Compared to the best statically chosen configuration for performance, dynamic allocation of cores decrease the System Vulnerability by as much as 37% with less than a 5% overhead. Alternately, compared to the best statically chosen configuration for vulnerability reduction, dynamic allocation of cores decrease the execution time by 60% while increasing the System Vulnerability by only 7%. This clearly emphasizes the ability of our schemes to provide good performance-reliability tradeoffs. Future extensions involve extending this study to heterogeneous multicore platforms and investigating the impact of consolidating multiple applications on the soft error vulnerability of the underlying system.

Chapter 5

Impact of Dynamic Voltage Frequency Scaling on the Architectural Vulnerability of GALS Architectures

5.1 Introduction

Ever-increasing power consumption is driving microarchitects to consider alternate design solutions. One popular solution is the Globally Asynchronous Locally Synchronous (GALS) design that incorporates multiple independent clocks driving different parts of the processor independently. A Multi-Clock Domain (MCD) design allows the different domains to be run at different voltage and frequency levels providing an ideal platform for applying Dynamic Voltage-Frequency Scaling (DVFS). Several runtime DVFS algorithms have been proposed in literature to increase the power efficiency of a MCD platform [109] .

In addition to power constraints, reliability concerns due to transient errors, as discussed earlier, have emerged as an important design constraint. Past works [115] have explored the impact of voltage scaling techniques on raw soft error rates (SER). In addition to the raw SER, the overall system SER also depends on the AVF, refer chapter 3. Since DVFS causes instruction flow changes through the

pipeline, it affects the utilization of the structures and thereby has an impact on AVF. Here we study the impact of applying different DVFS algorithms on AVF in a GALS environment. This would in turn help designers in choosing the optimal algorithm that meets specific reliability goals. Our current work explores AVF of the issue queue due to its importance in a superscalar design and because prior results show that issue queues have significant AVF [41]. To our knowledge, this is the first effort that explores the architectural vulnerability analysis of DVFS algorithms.

Our results show that except for one DVFS algorithm (Greedy Search [69]), the overall AVF increases when applying DVFS when compared with a Non-DVFS environment. Considering the fact that AVF is inversely proportional to the Mean Time To Failure (MTTF) of a system [75], applying DVFS could increase the failure rate of a system. Since the DVFS algorithms have varying performance-power tradeoffs, we characterize the AVF variations with respect to these parameters as well. Our study shows that the Non-DVFS case does better than any of the DVFS algorithms in terms of reducing AVF per IPC (AVF/IPC). In terms of reducing both total power and AVF ($TotalPower * AVF$), the Greedy search and PI [128] DVFS algorithms provide the best option. Combining all three parameters we look at characterizing an algorithm's *Vulnerability efficiency*, given in terms of $AVF{*}Watts/IPC$, which gives a DVFS algorithm's capability to reduce AVF and power without degrading performance significantly. We find that voltage-frequency scaling using DVFS does not provide good vulnerability efficiency as compared with a Non-DVFS environment. Amongst the DVFS algorithms, Attack-Decay [95] and PI provide good vulnerability efficiency.

Section 5.2 provides an introduction to Architectural Vulnerability Factor and how it impacts the overall MTTF. Section 5.3 discusses the different DVFS algorithms we analyze in our work while section 5.4 provides the different configuration parameters used in our study. Section 5.5 looks at the AVF characteristics of the DVFS algorithms. The section also analyzes the vulnerability efficiency of the DVFS algorithms.

5.2 Background

This section provides a background on the MCD platform used in this study.

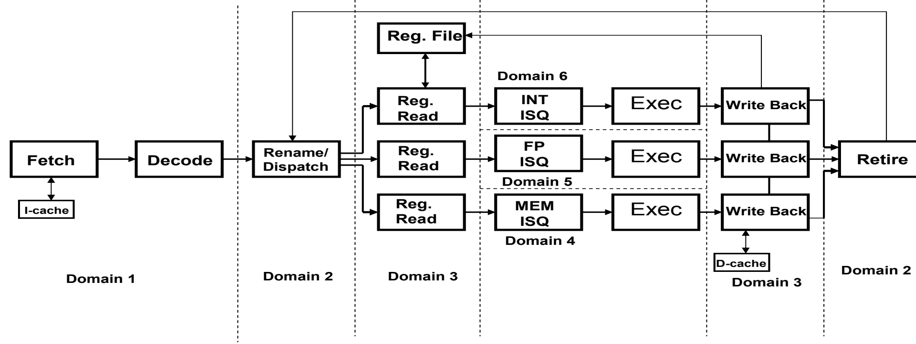


Figure 5.1. GALS pipeline showing the different asynchronous domains

The pipeline design is shown in Figure 5.1 [108, 109]. It includes 6 asynchronous domains which are the Fetch, Rename, Register Read, Integer (Int), Floating-Point (FP) and Memory (Mem) domains. Each of the domains operating at different frequencies interface with each other using asynchronous FIFOs [109]. The Fetch stage includes the Fetch queue, branch prediction, I-cache and the decode stage of pipeline. The Rename state includes register-renaming and the retire queue since every instruction retiring needs to update the register rename table. Register reading of source operands from the Int and FP register files occur in the Register Read stage after which the instruction move into the Int, FP or Mem domains based on their type. The Int, FP and Mem domains have separate issue queues besides execution logic. Since the address-generation to actual load-store instruction execution has a significant impact on performance [133], the Mem domain has a separate issue queue for issuing the effective address instructions. We perform sensitivity analysis of our evaluation by varying our baseline architecture in later sections.

5.3 DVFS Algorithms

As the importance and need for asynchronous designs have grown, so have the number of algorithms that optimize their power and performance. We analyze the

following representative set of DVFS algorithms in this work.

Threshold algorithm: Threshold is a simple algorithm that does the voltage-frequency (VF) switching based on pre-defined utilization thresholds for different structures across different domains [109]. For each interval, the utilization is evaluated as the number of instructions passing through the structure divided by the interval size. If it lies above a threshold, the voltage-level of that domain is increased while if it lies below a threshold, it is decreased. Since AVF is related to utilization of structures, lowering VF only when utilization is low could have minimal impact on AVF. The constraint of a fixed threshold affects the efficiency of the algorithm. A benchmark whose structure utilization is just below the threshold for decreasing VF could have a much higher AVF than a benchmark whose utilization is slightly above the threshold but runs at a higher VF level.

Attack-Decay (AD) algorithm: This algorithm [95] tracks the issue queue occupancy changes over two neighboring periods and whenever a rapid change is detected, it triggers a *attack* phase which corresponds to a rapid scaling in VF level by a fixed number of VF levels. In other cases, the VF level is decreased slowly but continuously as long as performance drop does not exceed pre-defined thresholds. This algorithm is applied only across domains 4, 5 and 6 (see Figure 5.1). By taking benchmark characteristics into account, AD does not suffer the problem of fixed thresholds. But taking decisions based on information from only two neighboring periods and allowing continuous decay (within certain bounds) might not have an optimal impact on overall AVF.

Modified Attack-Decay (ModAD) algorithm: The Modified Attack-Decay algorithm [133] was proposed to optimize on the attack phase of the original AD algorithm. In original AD, the attack phase involved changing the VF level by a fixed number of VF levels. But the fixed change leads to mismatch between the magnitude of queue occupancy change and VF change and could result in sub-optimal energy savings or performance. In ModAD, the attack phase is changed by making the number of VF levels to linearly change with respect to queue occupancy changes. Since AVF corresponds to queue occupancy, scaling VF level

linearly with respect to occupancy changes should enable ModAD to optimize a structure's AVF better than AD. But it faces the same problems as AD with respect to the performance degradation in the decay phase. Also sub-optimal VF changes due to intermittent pipeline phenomena (like L2 cache miss) could affect the AVF.

Greedy Search (Greedy) algorithm: Energy efficiency can be expressed in terms of Energy-Delay square product (ED^2). Greedy search [69] approximates the optimal ED^2 value based on the history of last two intervals. If the ED^2 was reduced over the two intervals, the VF change was correct and we optimize in same direction. Otherwise, if ED^2 was increased, the optimization direction is reversed. Each domain has a sample phase and hold phase. The sample phase is when a domain optimizes its VF level. In the hold phase, it operates in the same VF level and another domain does the sampling then. The sample phase for the domains goes in a round-robin fashion. Optimizing ED^2 which is a function of occupancy and time both of which affect AVF as well indicates that greedy search algorithm might probably optimize AVF the best. But the hold phase of each domain could lead to sub-optimal impact on AVF, since a domain operates in the same VF during the hold phase.

PI algorithm: PI [128] is an analytic online DVFS scheme based on modeling the queue-domain network. Issue queue occupancies are used as feedback signals to control the VF setting. The service rate over a specific interval is computed as

$$\mu_k = \mu_{k-1} + K_I(\bar{q_k'} - q_{ref}) + K_P(\bar{q_k'} - \bar{q'}_{k-1})$$

where μ refers to the service rate, $\bar{q_k'}$ refers to the queue occupancy over interval k, $\bar{q'}_{k-1}$ is the queue occupancy over interval *k-1*, q_{ref} is the threshold queue occupancy which is pre-determined. K_I and K_P are control parameters (for further details, refer [128]). The new frequency f_k is computed using the relation $\mu = IPC * f$. If fine-grained VF transitions are not possible, the VF setting is approximated to the closest available level. Since PI computes the service rate independent of IPC and since the algorithm dependents on the q_{ref} parameter,

it could affect the AVF of structures similar to the threshold algorithm. But PI tracks benchmark characteristics better than threshold since the queue occupancy over successive intervals is tracked.

Baseline Parameters

Parameter	Value
Pipeline Width	4
Branch-Predictor	2-level
Fetch Queue/RUU/LSQ	16/128/64
ISQ Int / FP / Mem	16 / 8 / 8
Int ALUs / Mult. / Div.	4(1-cycle latency) / 2(3) / 2(20)
FP ALUs / Mult. / Div.	2(2) / 2(4) / 2(12)
L1 D-Cache	64KB, 2-way 32B block, 2 ports (2)
L1 I-Cache	32KB, 2-way 32B block (2)
L2 Unified	512 KB, 4-way 32B line-size (12)
I-TLB / D-TLB	512-entries 4-way/ 1K-entries 4-way
Mem. Lat / Baseline Freq.	200 cycles/ 4.0 GHz
Technology / Voltages	45 nm / V_{DD} = 1.0 V, V_{TH} = 0.151 V
DVFS Interval Period	20,000 cycles (at baseline frequency)

DVFS Algorithm Parameters

Parameter	Value
Threshold	Int ISQ - [4,8] FP ISQ - [3,5] Mem ISQ - [3,5]
AD / ModAD	Attack Phase - 10% change in queue occupancy
Greedy Search	Sample Phase is 4 intervals
PI	K_I=0.6 K_P = 0.2, Int ISQ q_{ref}=6 entries, FP and Mem ISQ q_{ref}= 4 entries

Table 5.1. Simulation parameters.

5.4 Simulation Setup

Table 5.1 gives the simulation configuration and DVFS algorithm parameters used in our experiments. The MCD environment was based on the simulation model developed in [109]. The model uses Wattch [23] to compute both static and dynamic power. The framework was modified to compute the AVF of the different structures. The default configuration has the 6 asynchronous domains discussed in the pipeline model in section 5.2. Since we apply DVFS, arbiter-based asynchronous FIFOs are used for the asynchronous domains to interface [109]. All our experiments were done on 16 SPEC2000 benchmarks, 7 Int and 9 FP benchmarks, each run for 100 million instructions after fast-forwarding to an early SimPoint [98]. Parameters for the different algorithms were chosen after careful consideration based on empirical results.

Besides the DVFS algorithms, we also consider a Non-DVFS case, where all

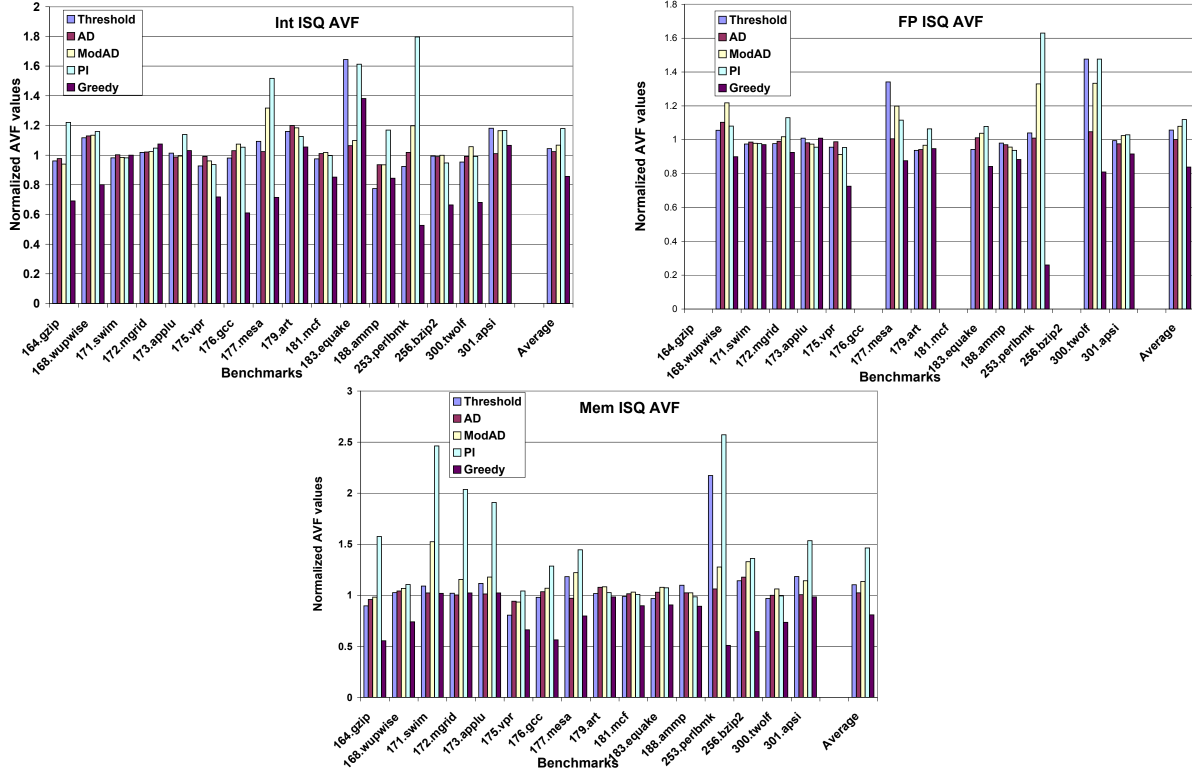


Figure 5.2. Impact of different DVFS algorithms on issue queue's AVF.

the domains are run at same frequency (4.0 GHz) through the entire simulation run. Note that we do not include a case of a single synchronous domain since the microarchitecture in that case would be different.

5.5 Experimental Results

In this section, we study the AVF characteristics of Int, FP and Memory issue queues with respect to the different DVFS algorithms. While previous works [51] have looked into the performance-power tradeoffs, we show that their AVF impact cannot be ignored when studying these algorithms. Since different DVFS algorithms have varying behavior, the performance-AVF and power-AVF tradeoffs involved with the different algorithms is analyzed as well. Since the raw error rate changes with voltage-scaling, we study the DVFS impact on the overall FIT rate. All the results presented in this section are normalized to the Non-DVFS case. Also some of the SPECInt benchmarks did not use the FP ISQ and hence had zero AVF. Hence these benchmarks appear with zero value in the FP ISQ graphs.

5.5.1 AVF Variation across DVFS algorithms

Figure 5.2 shows the percentage AVF variation across the different DVFS algorithms for the Int, FP and Mem issue queues. Across the different DVFS algorithms, the AVF of structures could vary by as much as 80% (seen in Mem ISQ AVFs when applying Greedy and PI algorithms) showing that choosing the right DVFS algorithm can have significant impact on the MTTF. Further we notice that, on an average, only the Greedy algorithm does better than a Non-DVFS case. The

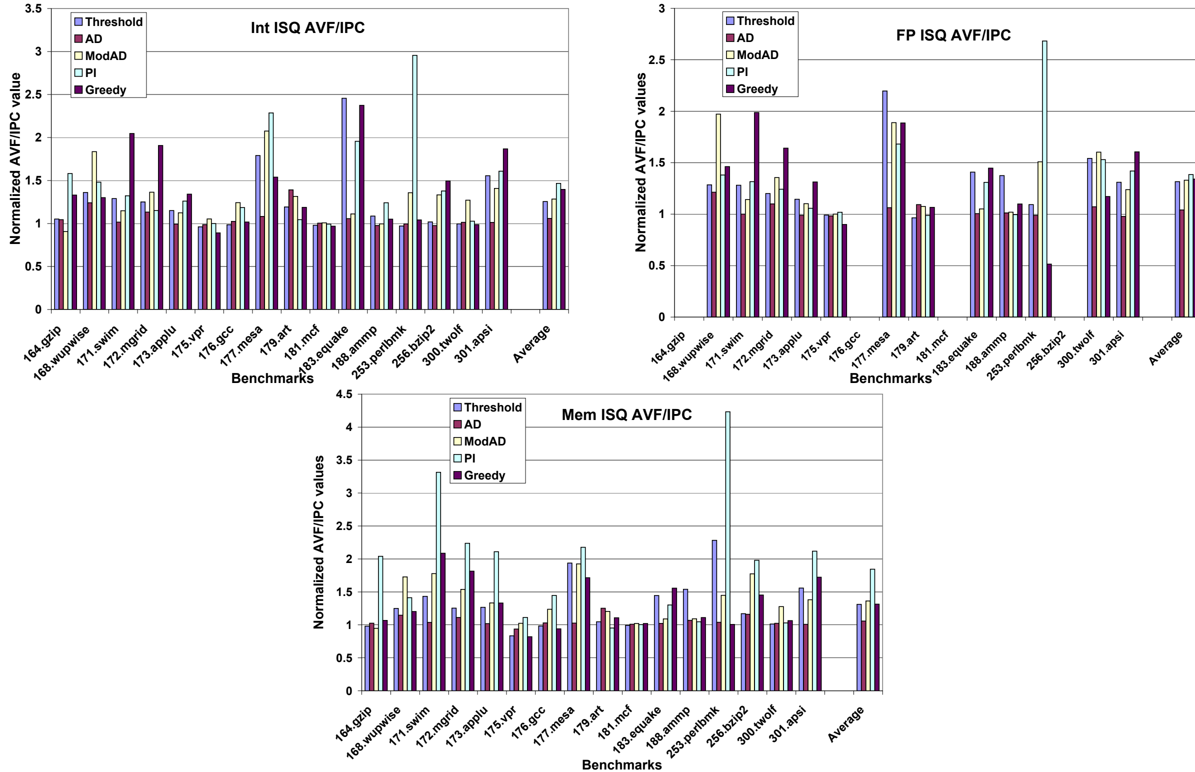


Figure 5.3. Performance Vulnerability tradeoffs of the DVFS algorithms.

Greedy criterion of optimizing on energy and delay have significant correlation to issue queue occupancies and thereby reduces the AVF of the structures. PI causes the biggest increase in AVF of the issue queues because inter-domain dependencies cause occupancy increase in a dependent domain while the VF level in the source domain does not scale correspondingly. Interestingly ModAD increases AVF of issue queues more than AD. This is because inter-domain dependencies cause the fixed VF level change in the attack phase of AD to recover better than ModAD.

For example, in 177.mesa the decay phase causes the memory domain (incremental increase in entries and bursty issue) to operate at lowest VF-level while the Int and FP domain are dependent on its execution.

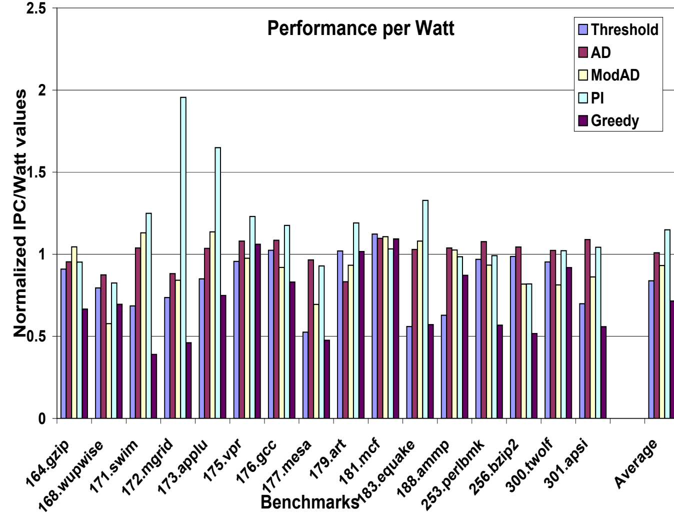


Figure 5.4. Performance per Watt of the DVFS algorithms.

5.5.2 Performance-Power tradeoffs

Previous works have studied the performance-power tradeoffs when applying DVFS. We summarize the results here and use it to contrast against those which incorporate AVF variations as well. Figure 5.4 shows the results. We see that the PI and AD algorithms do well in terms of increasing the performance/watt compared to the Non-DVFS case.

5.5.3 Performance-Vulnerability tradeoffs

A DVFS algorithm could do well in terms of reducing AVF of a structure but it might have a big performance impact as well. This would mean that entries occupy the structure longer and hence are more vulnerable to soft errors compared to another algorithm which has high AVF but negligible performance impact. Hence we look at Vulnerability per IPC (AVF/IPC) in Figure 5.3.The results show that all the DVFS algorithms have worse AVF/IPC than Non-DVFS. The Greedy algorithm which had the smallest AVF now has pretty high AVF/IPC values due to the larger performance degradation. For algorithms like AD and ModAD, even

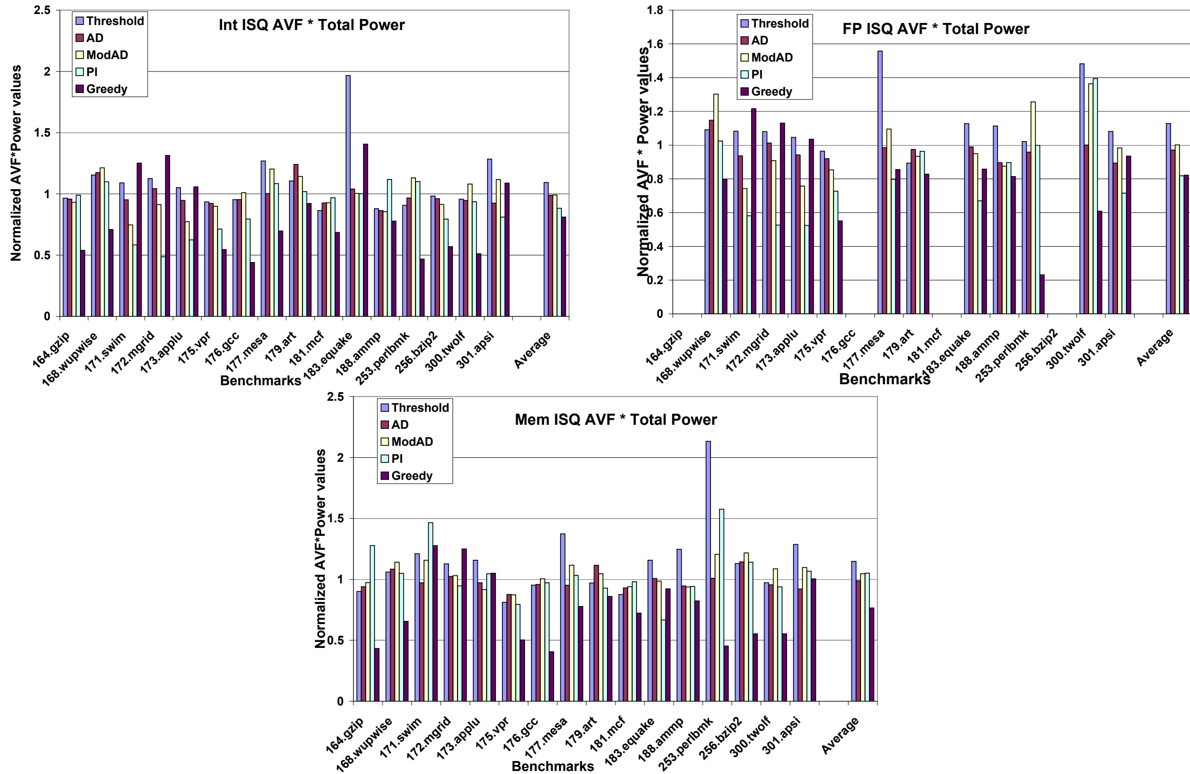


Figure 5.5. Power Vulnerability tradeoffs of the different DVFS algorithms.

though IPC is a VF scaling criterion, the performance degradation that is allowed between intervals (during the *Decay* phase) causes higher AVF/IPC.

5.5.4 Power-Vulnerability optimization

Though high performance increase is a primary goal, power and reliability (in terms of AVF) can no longer be ignored in current day microprocessors. Figure 5.5 shows the ability of DVFS algorithms to reduce both power and AVF ($Total\ Power * AVF$). Both the Greedy and PI algorithms do well in terms of reducing power and AVF mainly due to their ability to reduce power consumption.

5.5.5 Vulnerability Efficiency characterization

Vulnerability Efficiency, given by $AVF * Total\ Power/IPC$, characterizes the ability of a DVFS algorithm to reduce AVF and power and also have a minimal impact on performance degradation. Given the importance of reliable operation in

system design, merely looking at performance per watt goals for choosing DVFS algorithms need not provide an optimal operating point. Figure 5.6 shows the vulnerability efficiency of the different DVFS algorithms is worse than Non-DVFS for Int, FP and Mem issue queues. For the DVFS algorithms, we clearly see that no single algorithm does consistently better than others though AD and PI provide the best average case vulnerability efficiency.

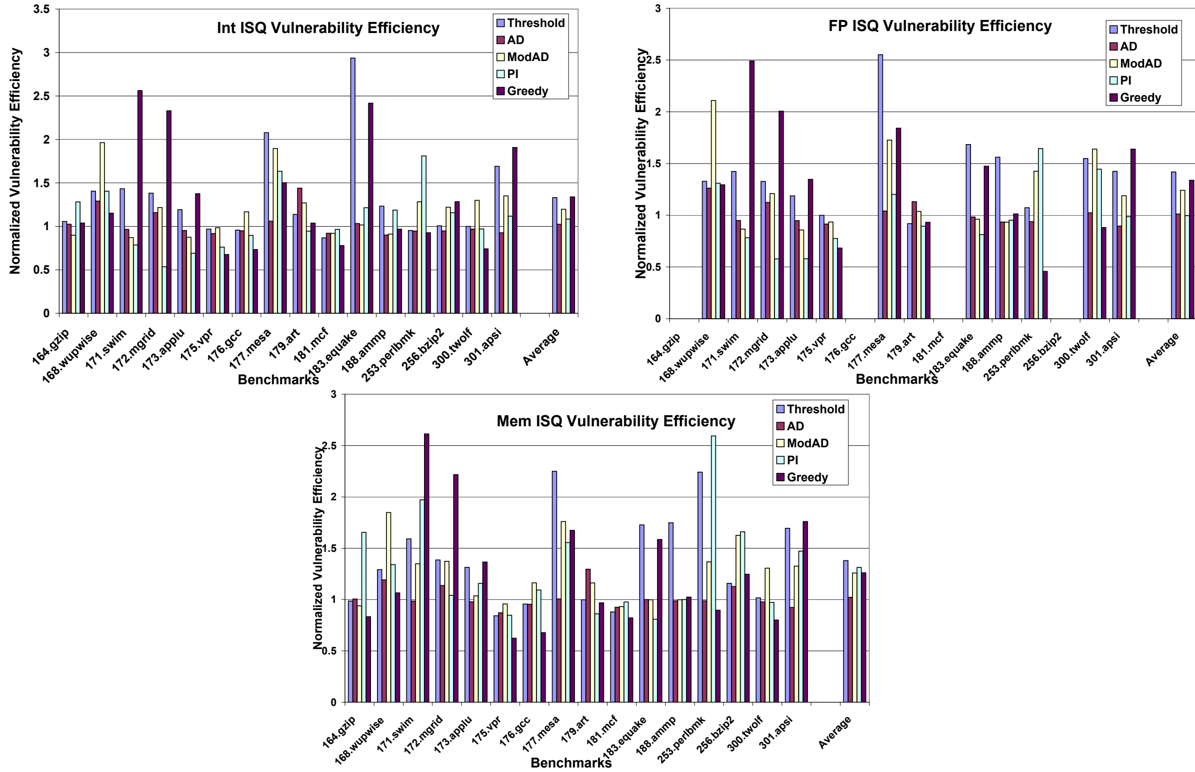


Figure 5.6. Vulnerability Efficiency of the different DVFS algorithms.

5.5.6 Impact of voltage scaling on overall FIT rate

Works like [115, 99] have pointed out that the raw FIT rate increases with decreasing voltages. Since DVFS algorithms scale voltages based on different criteria, they affect the raw FIT rates during those periods. Raw FIT rate variation with voltage is given by $FIT_{raw} = K.exp(-Q_{crit}/Q_s)$ where K is a constant for a particular technology generation, Q_{crit} is the critical charge for a SRAM cell and Q_s is charge collection efficiency. Since both Q_{crit} and Q_s change when the sup-

ply voltage is scaled, the relationship between raw error rate and voltage is not straight-forward. [115] shows empirical results which indicates that there exists a super-linear relationship between voltage decrease and SER increase. Here we assume both a linear and exponential relation between voltage change and raw SER and compute the overall FIT rate. Figure 5.9 shows the overall FIT rate averaged

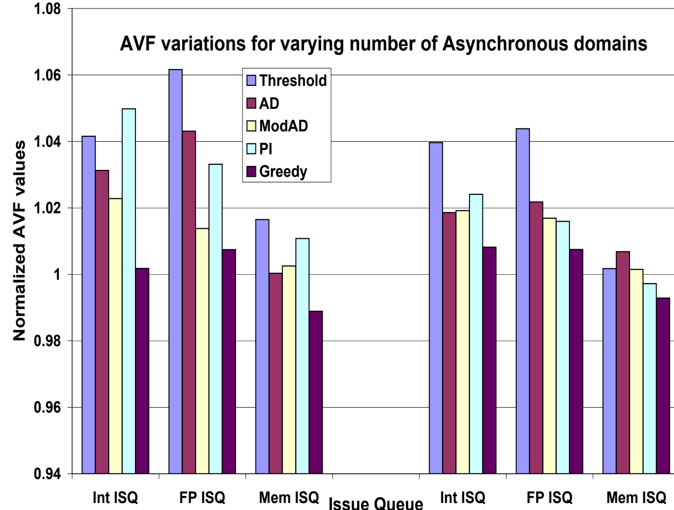


Figure 5.7. Average AVF variation for GALS pipeline with 4 and 5 asynchronous domains

over all benchmarks. We see that the Greedy algorithm has the lowest overall SER followed by the Non-DVFS case. Looking at Figure 5.9(A) which gives the time fraction spent in different voltage levels some interesting observations can be made. Since DVFS algorithms typically reduce the voltage level when occupancy reduces, we anticipate the AVF to be lower. However, lower voltages increase the raw SER. This trend where the raw SER dominates AVF in determining the overall FIT rate is seen when the threshold algorithm is run for the FP issue queues. But consider the Greedy algorithm and Non-DVFS case for Int issue queues. In this case, even though the Non-DVFS case spends its entire execution time in the highest VF level (reduced raw error rates), it has higher overall SER due to the AVF factor dominating raw error rates.

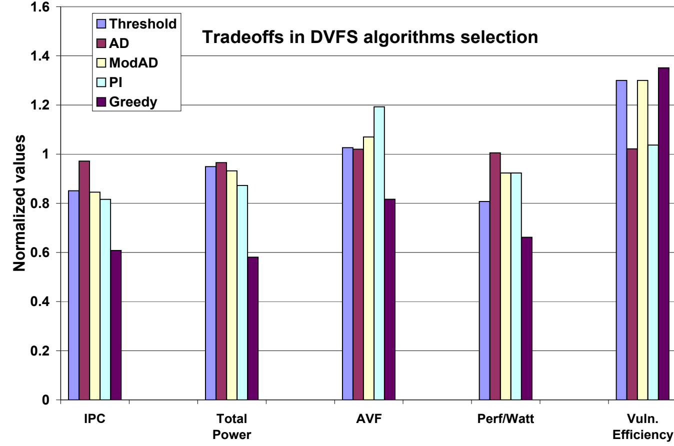


Figure 5.8. Performance, Power and Reliability tradeoffs provided by different DVFS algorithms

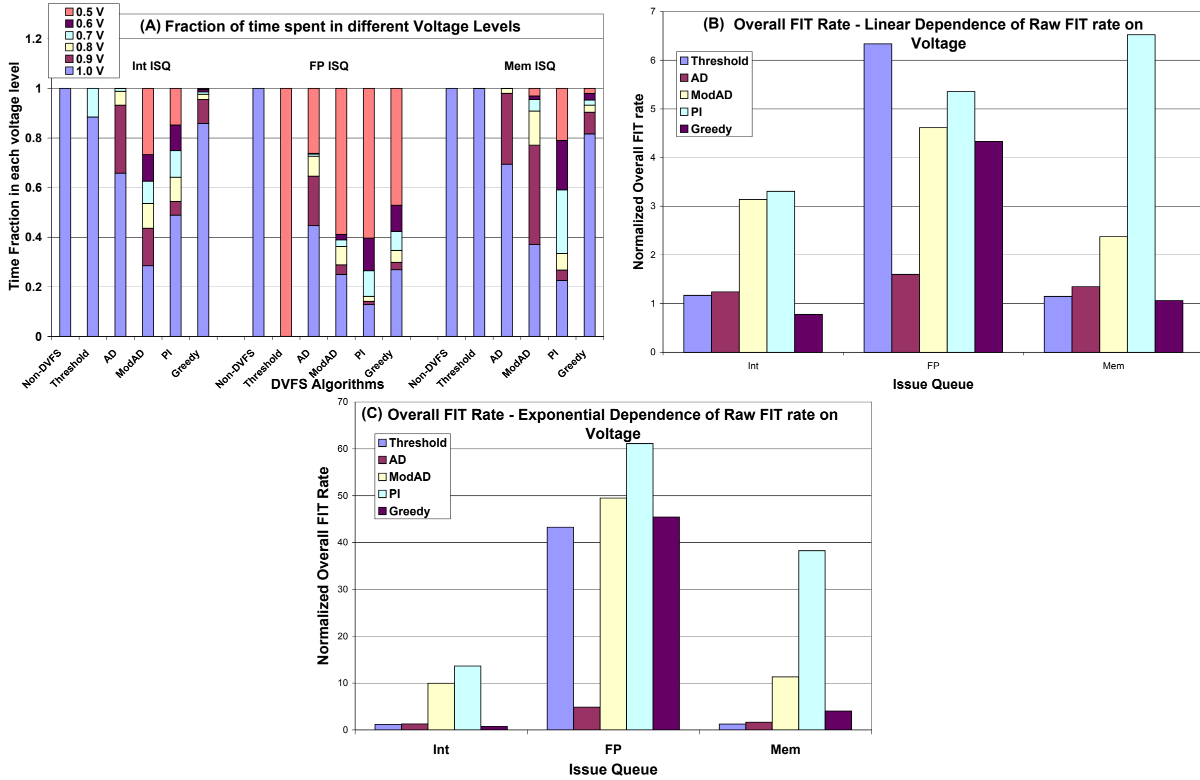


Figure 5.9. (A) Fraction of time each DVFS algorithm spends in a VF level. (B) Linear raw error rate voltage relationship. (C) Exponential raw error rate voltage relationship.

5.5.7 Sensitivity Analysis

Here we look at the impact of varying the number of asynchronous domains (4 and 5 asynchronous domains) on the AVF of the issue queues when applying the

different DVFS algorithms. Figure 5.7 shows the average percentage increase in AVF across all benchmarks for the three issue queues. We find that the Non-DVFS case has the lowest AVF for Int and FP issue queues while for Mem issue queues the Greedy Search algorithm optimizes best. In general, the AVF variation between algorithms decreases with decrease in number of domains. But this also reduces the efficiency of the algorithms in applying DVFS as more domains are brought together.

5.6 Summary

Conventionally DVFS algorithms are studied based on their performance per watt, without considering their reliability impact. Growing concern for soft errors and the impact of DVFS algorithms on instruction occupancy in pipeline structures necessitated us to look at their AVF characteristics. Figure 5.8 shows the IPC, total power and AVF variations of the DVFS algorithms along with their performance per watt and vulnerability efficiency. We see that the DVFS algorithms have significantly different AVFs, thereby impacting the overall MTTF of the system. Classifying DVFS algorithms based on their vulnerability efficiency is important to meet reliability goals and such a classification leads to very different algorithm choices compared to ones based only on performance and power. As a future work, the analysis needs to be extended to look at the entire system.

Chapter 6

Mechanisms to address Non-uniform aging

6.1 Introduction

The emergence of phenomena such as Time Dependent Dielectric Breakdown (TDDB), Negative Bias Temperature Instability (NBTI), Hot Carrier Effects (HCE) and Electro-Migration (EM) have all contributed towards decreasing circuit reliability by progressively slowing down transistors during their normal operation. This gradual slowdown in circuit behavior, commonly known as aging or wearout, can eventually result in circuit failures by causing hard faults and timing failures. As technology continues to scale well into the nanoscale regime, aging-induced timing failures are expected to become increasingly prevalent, resulting in reduced microprocessor lifetimes [18]. While the turn-around rate for personal electronic systems is often less than 3 years, processors in data centers, automotives, military/space equipment, and embedded systems remain functional for long periods ranging from 10 to 25 years [93]. Gartner research conducted a survey, mentioned in [15], which showed a significant percentage of organizations (about 50%) running systems that are more than 10 years old. The requirement for longevity in these systems stem from a variety of reasons, including them being legacy systems, high cost of replacement/alterations and/or non-accessibility of the system after deployment. Mitigating the impact of aging hence is very important in these

market segments.

Aging is a result of thinning gate-oxides and non-ideal voltage scaling in transistors [105]. The rate of aging is determined by a collection of factors including supply voltage, temperature, switching activity, gate bias, along with static and leakage currents. Two major aging phenomena that will effect transistor operating speeds in upcoming technology nodes are Negative Bias Temperature Instability (NBTI) and Hot Carrier Effects (HCE) [45]. Both NBTI and HCE have the potential to increase the threshold voltage (V_{th}) of the pMOS and nMOS devices respectively, resulting in a reduction of circuit speeds by more than 20% [121].

Conventional microprocessor pipeline incorporates numerous storage sub-components that are critical for high-performance operation. In particular, *collapsing queue-based* issue queues are vulnerable to non-uniform wearout across its entries. Collapsing queues, which are typically implemented using latch-based designs [39], are essential for high-speed issue queues because they simplify the associated selection logic [25]. Modern processors such as AMD Phenom and Opteron processors employ collapsing queues that are latch-based. The latch-based collapsing queues are legacy designs and adding incremental improvements such as clock-gating involves less verification than completely moving to other recently proposed low-power designs [25].

The degree of aging depends on different factors, which includes the clock frequency, number of writes, the data value being written and the switching activity. Also the correlation between these factors often leads to conflicting objectives when trying to reduce aging. For example, higher write activity often implies more diversity of the values stored in a particular entry (leading to reduced wearout for a subset of transistors due to NBTI). At the same time, this higher activity increases wearout in another subset of transistors due to HCE. Therefore, it is imperative that the proposed solution take into consideration these tradeoffs resulting from the different interacting aging phenomena when looking to reduce their impact.

A conventional solution for ensuring that timing requirements are met is frequency guardbanding, where designers add a timing slack in order to account for small shifts in frequency over the devices' lifetime. However, as the impact of the aging mechanisms increase, conservative guardbanding will significantly erode the benefits achieved by moving to a lower technology. As a result, aging aware de-

signs become vital for upcoming technology generations. Past solutions, such as periodic data inversion [61] and core-level scheduling [78] [107], propose schemes to mitigate the effects of aging. Also some of these techniques require continuous sampling of the data bias in the cells, aside from the additional hardware to provide the required bit inversion, to mitigate wearout [2]. While these techniques are effective, the continuous sampling adds a non-negligible power overhead, especially since aging is a long term phenomenon, under different operating conditions. Also these approaches have been limited to a single wearout mechanism and do not look at decreasing the switching activity in the structures which significantly impacts the aging degradation. The restricted collapsing proposed here does not involve any monitoring of the cell bias in the structures (and hence no sampling hardware) but operates on all entries to reduce the variation in switching activity. Besides making the aging rate more uniform, these mechanisms also reduce the worst case degradation thereby increasing the lifetime of individual cells. This in turn improves the overall throughput of the system given that the issue queue is a key component in determining the operating frequency.

The major contributions of this work are:

- Studying the impact of NBTI and HCE on collapsible issue queues with non-uniform aging. The results show that aging-unaware designs can have switching activity that varies up to 38% for the issue queue.

- Aging analysis performed across the entire SPEC2K suite at 32nm technology. In the collapsing queue-based issue queue, we estimate the increase in read delay. The mitigation technique decreases the worst-case cell read delay for collapsing queues by 32% on average.

The remainder of the chapter proceeds as follows. Section 6.2 provides a background on NBTI and HCE and describes the pipeline microarchitecture used in this work. Section 6.3 illustrates the switching activity variability across entries for the issue queue. Section 6.4 provides the experimental setup. Section 6.5 discusses the proposed mechanism to reduce issue queue degradation and discusses the impact on the read delay degradation. Section 6.6 discusses related work and finally section 6.7 summarizes this chapter.

6.2 Background

This section provides a background on NBTI and HCE degradation. We also discuss the baseline microarchitecture we use for our analysis.

6.2.1 Impact of NBTI and HCE

NBTI is a result of the continuous generation of traps at the Si/SiO_2 interface of a *p*MOS transistor when its gate voltage is negative (input is "0"). Removal of the stress (input is "1") helps in partial recovery by annealing a portion of the traps generated. The recovery is only partial and hence, over time, it leads to an increase in the threshold voltage (V_{th}) of the transistor. The modeling of NBTI in this work is based upon the long term prediction model proposed in Bhardwaj et al. [13]. The change in threshold voltage resulting from NBTI is modeled by the expression in Equation 1:

$$\Delta V_{th} = \left(\frac{\sqrt{K_v^2 \cdot T_{clk} \cdot \alpha / min(\alpha, 1-\alpha)}}{1 - \beta_t^{1/2n}} \right)^{2n} \tag{6.1}$$

The change in V_{th} due to NBTI stress is represented as a function of the electric field, the oxide thickness, temperature, and oxide capacitance (K_v), the duty cycle (α), and the switching frequency (T_{clk}). NBTI recovery is represented by the denominator with β_t capturing the dependence of fractional recovery on the oxide thickness, t_{ox}. The closed-form upper bound for long term NBTI degradation was found to be within 0.1% of NBTI degradation estimation using separate stress and recovery cycles [13].

HCE is a phenomenon that causes defects at the SiO_2 interfaces within transistors. The defects can be attributed to carrier (electron or holes depending on *n*MOS or *p*MOS) heating in the high electric field near the drain side of a transistor and ultimately results in the shift of device parameters. Traditionally, HCE has been more significant in *n*MOS transistors because electrons have higher mobilities than holes and gain higher energy from the channel electric field [60]. HCE degradation occurs during the low-to-high transition of the *n*MOS transistor and undergoes negligible recovery. As a result, HCE degradation is highly dependent

upon the amount of switching activity. Similar to NBTI, HCE causes an increase in the threshold voltage of the nMOS transistor, reducing a circuit's performance over time. The modeling of HCE in this work is based upon the predictive model of Wang et al. [120]. The change in threshold voltage resulting from HCE is modeled by the expression in Equation 2:

$$\Delta V_{th} = \frac{q}{C_{ox}} K_2 \sqrt{Q_i} exp\left(\frac{E_{ox}}{E_{o2}}\right) exp\left(-\frac{\varphi_{it}}{q\lambda E_m}\right) t^{n'} \tag{6.2}$$

The change in V_{th} due to HCE is most importantly a function of the electric field (E_{ox}), inversion charge (Q_i), and a function based on temperature (K_2). Please refer [120] for more information on the other parameters. The amount of time a particular transistor undergoes stress through its lifetime is expressed through t with a time exponential constant (n') of 0.45. This model, along with the constants used for fitting it to experimental data, have been verified in [120].

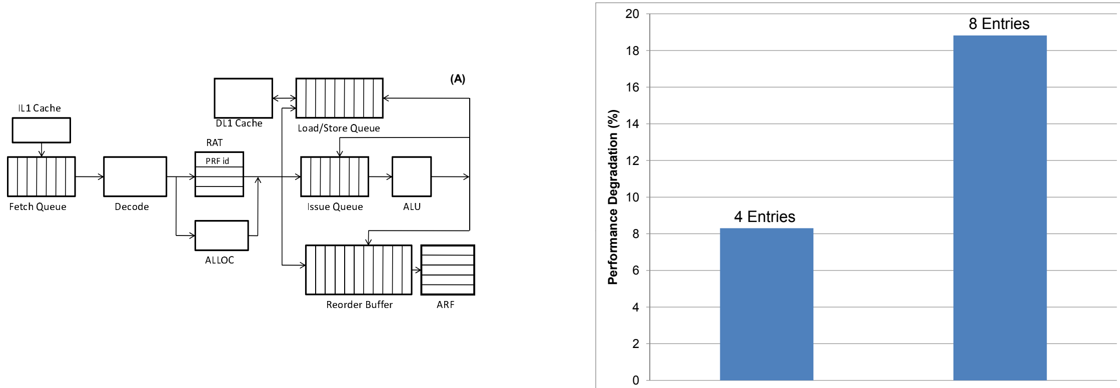


Figure 6.1. (A) Base pipeline we use in this study. (B) Performance loss due to shutting down entries shown for a SPEC2K benchmark(*art*)

6.2.2 Baseline Microarchitecture

Figure 6.1A shows the pipeline considered in this work, which is similar to a P6-style pipeline [97]. Instructions fetched from the I-cache are decoded, in-order, in the front-end of the pipeline. Once decoded, the Register Alias Table (RAT) does the renaming of the logical registers by providing tags corresponding to the source and destination registers. These tags are Reorder Buffer (ROB) entries which also correspond to the Physical Register File (PRF) ids. The ROB enables in-order commit of instructions from the pipeline. The Architectural Register File (ARF)

exists as a separate structure in which the non-speculative instructions retiring from the ROB write their results. The Issue Queue (ISQ) issues instructions out-of-order to capture the instruction level parallelism (ILP) available in the application. The Load Store Queue (LSQ) buffers the load and store instructions before issuing them to memory. The LSQ provides memory-address disambiguation between store-load instructions, store to load data-forwarding, and in-order issue of stores to memory. The Alloc logic assigns entries in the ISQ, ROB and LSQ once the instructions get decoded. The memory sub-system includes the L1 and L2 caches besides main memory.

6.3 Motivation

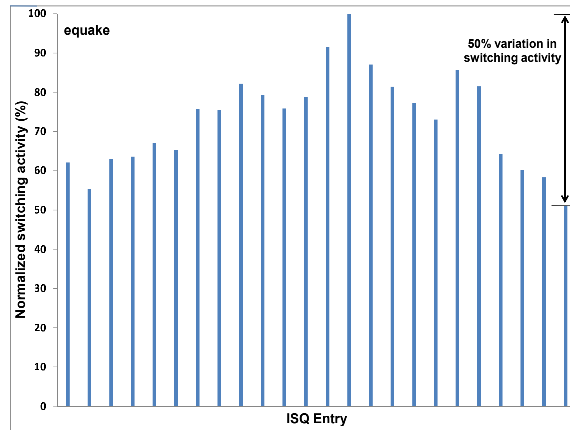


Figure 6.2. Switching activity variation across ISQ.

Switching activity plays an important role in HCE and the NBTI degradation. While NBTI is directly dependent on stored data value, very low switching in an entry affects the data bias in cells constituting these entries and aggravates NBTI degradation. Collapsible issue queue exhibit significant variation in switching activity across their entries. Figure 6.2 shows the variation in switching activity in the conventional issue queue. The significant variation in switching activity, as high as 50%, across the entries leads to non-uniform aging whereby certain entries become unfit to operate much earlier than others.Therefore, amortizing the overall switching activity across all entries in the structures has the ability to address the

HCE and NBTI degradation dominant in critical pipeline structures and this is what this work would looks at implementing.

One option to address aging is to shutdown entries that degrade beyond the allowed margin rather than trying to mitigate aging effects. However, this option can result in significant performance degradation. For a 24-entry ISQ, the performance loss was about 8% on losing 4 entries. Consequently, this work proposes the microarchitectural design to enable restricted collapsing in the issue queue to decrease the variation in aging across the issue queue entries. The impact of the design changes on the *worst case* degrading cell has been evaluated and is shown that there are significant gains to adopting such a design.

6.4 Simulation setup

Baseline Parameters

Parameter	**Value**
Fetch/Decode/Issue/Commit Width	6
Fetch Queue	128 entries
Branch-Predictor	Combined Predictor
BTB / RAS	2K-entry 4-way / 64
RUU / LSQ / ISQ	96 / 32 /24 entries
Integer ALUs	6 (1-cycle latency)
Integer Multipliers/Dividers	4 (3,20)
FP ALUs	6 (2)
FP Mult./Div./Sqrt.	4 (4,12,24)
L1 D-Cache Ports	2
L1 D-Cache/I-Cache	64KB, 512 sets, 4-way 32B block (2)
L2 Unified	512 KB, 4-way 32B block (12)
I-TLB / D-TLB	512-entries 4-way / 1K-entries 4-way
TLB Miss-Latency	30 cycles
Memory Latency	150 cycles

Table 6.1. Simulation parameters. Latencies of ALUs/caches are given in parentheses. All ALU operations are pipelined except division and square-root

Architectural experiments were conducted in an execution-driven simulation environment, using the SimpleScalar 3.0 toolset [24]. The simulator was modified to track actual values stored in the entries during execution, enabling us to track the different parameters required to compute degradation. The proposed techniques were evaluated using all 26 benchmarks from the SPEC CPU2000 benchmark suite. The benchmarks were compiled for the Alpha ISA, and reference input sets were used. Results were obtained from a detailed simulation phase of 100 million instructions after fast-forwarding to the single SimPoint [98] of each

benchmark. The parameters of our baseline model are shown in Table 6.1.

To emphasize the ability of restricted collapsing to reduce the switching activity, we show the average reduction in switching activity across entries (averaged over all cells in an entry) in the issue queue. The degradation analysis, though, was done at a *per cell* granularity and all the degradation graphs report the *WORST CASE* degradation. We did not report the worst-case switching activity since it does not necessarily correspond to the cell with worst-case degradation, will discuss more in section 6.5, while the average value gives an indication of evening in read, write and data-switching activities.

All circuit level simulations were performed using HSPICE along with the Predictive Technology Model (PTM) [132]. Each structure was assumed to be implemented at 32nm technology node with nominal operating values verified against industry data. Since the issue queues are a temperature hotspot [101], the analysis is done at 110 °C.

For device degradation, each transistor in a given structure was assumed to degrade according to either Equation 1 (pMOS) or Equation 2 (nMOS). The values needed to determine a particular transistor's degradation were provided by the statistics gathered through architectural simulation. From the statistics, the 10-year degradation was then extracted for each transistor in the structure, important for systems with high availability requirements [93], by calculating the corresponding stress times. Once the transistor degradation was found, the BSIM4 model was updated with the result for each transistor's V_{th} shift and HSPICE simulations were performed to extract timing information. Also for SRAM-based structures, the SNM degradation was calculated. Since the working of the NMOS and PMOS transistor are inter-twined, we look at the overall degradation capturing the net effect of both phenomena. Also the degradation graphs report the standard deviation of the degradation across all the cells in the structure, shown as *Dev* in the figures. A smaller value of *Dev* emphasizes that more uniform aging in the cells within the different structures.

6.5 Mitigating switching activity variation in collapsible issue queues

Collapsing queues have high switching activity in their entries. Also there is significant variation in switching activity across different entries. Pipeline structures like the issue queue and load-store queues are implemented as collapsing queues. The AMD Opteron and Phenom processors use latch-based collapsing queues. Typically, latch-based collapsing queue design consist of inverters and transmission gates [39].

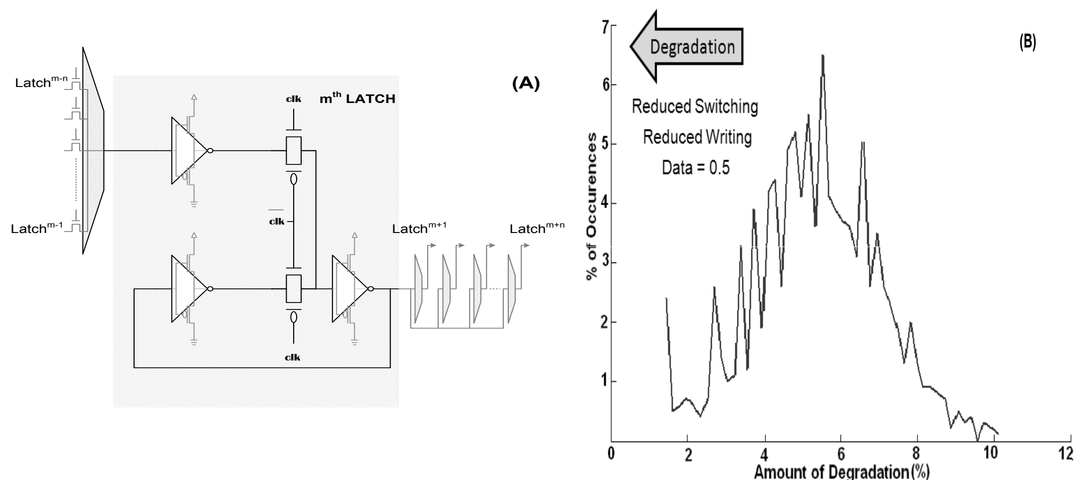


Figure 6.3. (A) Cell design in latch-based collapsible queues. (B) Distribution showing collapsing queue entry degradation as a function of data, switching activity and writes.

In collapsing queue structures, NBTI and HCE affect the delay from input to output and the minimum clock period for latching. For the collapsing queue structures, an entry being shown in Figure 6.3A, the input-output delay (data coming from Latch $m-1$ might be immediately read for execution) is the most critical. The amount of degradation due to NBTI is dependent upon the the data being stored in the entry while the amount of degradation due to HCE for a particular nMOS depends on the data being stored together with the data switching activity, read and write activity to that entry.

The combined effect of the two degradation mechanisms can have a significant effect on the delay through the structure as shown in Figure 6.3B. This shows the degradation distribution for a collapsing queue entry for various combinations of

data values, data switching activity, and writes. Note that the degradation values are not a linear combination of the different factors. Hence decreasing any one of the factors alone does not guarantee reduction in the degradation. The arrow indicates that a combination of decreased write activity and switching in a cell while maintaining the *BitOne Probability* around 0.50 (shown as $Data = 0.5$ in the figure) reduces degradation. Our proposed scheme achieves this by decreasing the collapsing activity, significantly reducing reads and writes activity variation, besides enabling the switching activity across all entries to be more uniform.

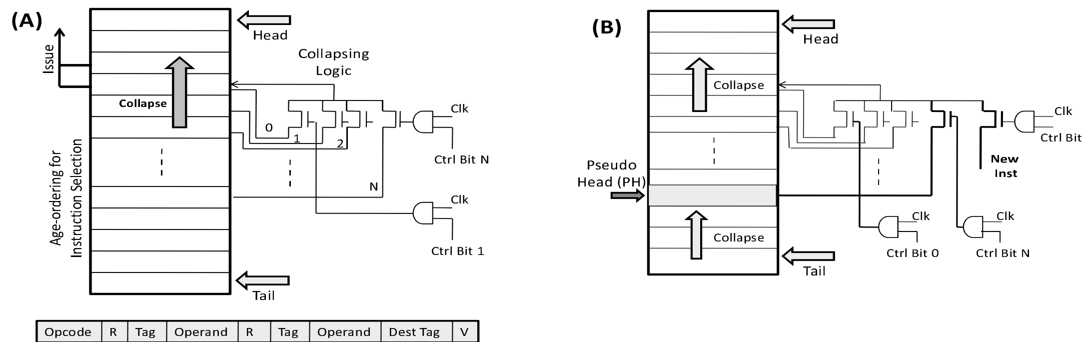


Figure 6.4. (A) Conventional collapsing logic design. (B) Modifications to support restricted collapsing.

6.5.0.1 Design for restricted collapsing in Issue Queues

In collapsing queue-based issue queues [39], an instruction issue from an entry causes all instructions in later entries to move forward. New instructions are always added to the *Tail* of the queue, which is usually the last few entries of the queue (varies based on the issue width). Such an implementation provides implicit age-ordering required by the selection logic to prioritize instructions for issue. While alternate implementations for issue queue design exists, these come with significant performance degradation [25]. Effective clock-gating schemes [62] bring down power requirements of the collapsing queues making them an attractive option due to the performance benefits they offer.

Figure 6.4A shows the conventional collapsing issue queue design. Entry movement between entries is done using pass-transistor MUXes, shown as *Collapsing Logic* in the figure. [39] discusses the algorithm for deciding when and how entries

get collapsed. The control bit, *Ctrl Bit*, is set for only one of the entries and the corresponding instruction moves into this entry. Every cycle, instruction movement into an entry is decided from the current entry and N other entries below this one, where N is the width of the pipeline. This movement between N neighboring entries is enough to allow new instructions to be added to the *Tail* of the queue and hence maintain the oldest-to-first instruction order.

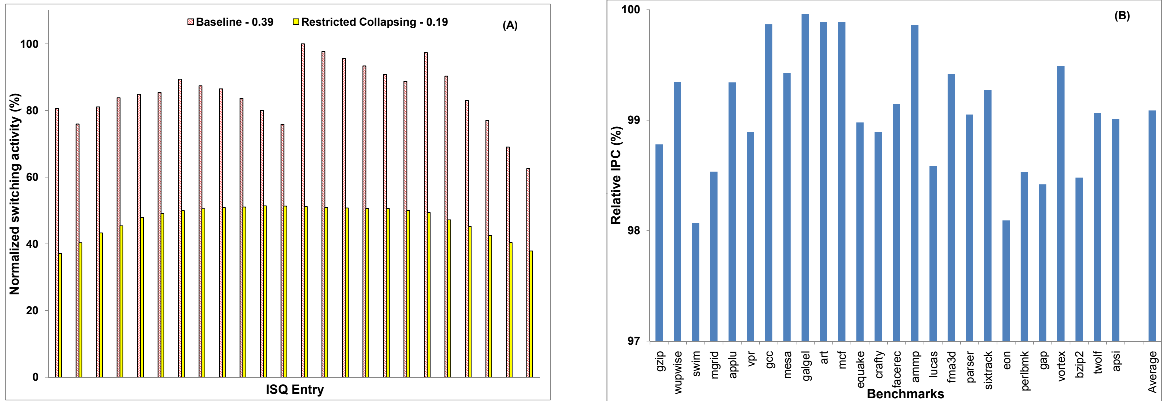


Figure 6.5. (A)Switching activity variation across ISQ entries. (B)Performance impact of Restricted Collapsing.

Collapsing and out-of-order issue from any entry means that the issue queue has significant switching activity combined with variation in the switching rate across entries. Figure 6.5A shows that, on average, the variation in switching could be as high as 38% between entries. As we saw in section 6.3, shutting down the wear-out affected entries comes with a performance degradation.

In the conventional design, instruction movement is controlled by the control bits. By resetting the control bits, instructions can be prevented from continuously moving towards the head of the queue. However, preventing any instruction movement within the issue queue reduces the performance gain that age-based ordering can provide. Therefore entry collapsing at any point of time is divided into two halves on either side of a particular entry, called the *pseudoHead* as shown in Figure 6.4B. This provides *restricted collapsing*, instructions in entries before the *pseudoHead* move towards the *Head* of the queue while those after the *pseudoHead* collapse only up to the *pseudoHead*. The restricted collapsing makes sure that the advantages of age-based ordering is not lost since most entries preserve the

relative ordering of instructions. Periodically, the *pseudoHead* is shifted amongst the different entries. In our simulations, we noticed that changing the *pseudoHead* every 1 million cycles was effective in balancing the collapsing activity.

Restricted collapsing is advantageous because collapsing instructions towards the head of the queue is not always required. During periods of low queue occupancy the age-ordering is maintained even when instructions occupy entries in the middle of queue. When dependency across instructions is high, implicit ordering is brought in. Complexity of flushing instructions from the issue queue due to a branch misprediction or other events is not increased since these events typically use the ROB tags, shown as *Dest Tag* in Figure 6.4A, to identify the specific entries to flush (since instructions issue out of order from the issue queue). By periodically varying the *pseudoHead*, the switching activity becomes more uniform across all entries.

Reducing the variation in switching between entries also has an effect on the switching activity of individual cells. For example, in the conventional design, since instructions are added in the last few entries, the cells corresponding to the valid bit in these entries are mostly set. If the entries in which new instructions are added get varied periodically, the bias in valid bit cells of the entries is reduced.

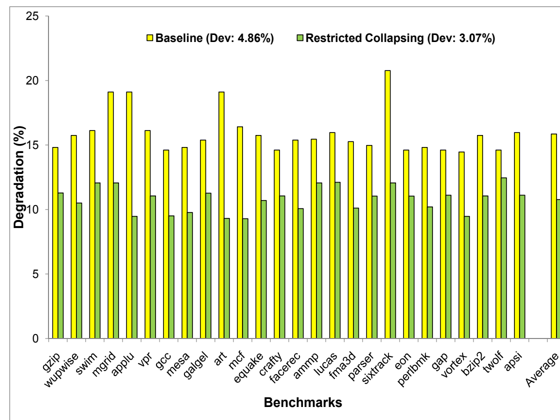


Figure 6.6. ISQ read delay degradation.

Also note that the control bits for entries after the *pseudoHead* are reset and cannot push instructions into earlier entries. The *pseudoHead* movement can affect the age-based ordering of instructions since entry availability can lead to instruc-

tions being distributed either side of the *pseudoHead*. The performance impact is shown in Figure 6.5B. Looking at the relative IPC, restricted collapsing respect to baseline, the overhead is below 1% on average. Figure 6.5A shows that the restricted collapsing scheme reduces the average switching activity across ISQ entries by 49% compared to the baseline scheme (from 0.39 to 0.19). Also the variation in switching activity between entries decreases to 14%.

Swapping source operands Sub-component analysis of issue queue entries (shown in Figure 6.4A) revealed that the higher-order bits of source operands have relatively higher zero probability that the other sub-components and hence degrade more. Their switching rate is much less than the rest of the entry. To counter this, we switch the upper half of an operand with the lower half. The switching is turned on/off periodically along with the *pseudoHead* update for the entire queue. To enable this swapping of operands, transmission-gate MUXes which have less than 2% timing overhead and 1% in area overhead (since the whole queue operates in a particular mode and doesn't require MUXes on a per-entry basis) are used. As instructions enter the queue, the operands get swapped and are reswapped again on issue to the ALU. Looking at the worst-case read delay degradations, in Figure 6.6, we see that restricted collapsing brings down degradation by 32.1% with respect to the baseline case. The figure also shows that the *Dev* reduces by 37% (from 4.86% to 3.07%) with respect to baseline enabling the issue queue to age more uniformly.

6.6 Related Work

As transistor integration increases, phenomena such as Electro-migration (EM), Time Dependent Dielectric Breakdown (TDDB), Hot Carrier Effects (HCE), Negative Bias Temperature Instability (NBTI) all impact the operation of transistors in the pipeline. Due to their differential impact on transistors - spatial and temporal, components can age faster or slower with respect to others. Modeling and studying these phenomena is important in terms of understanding their impact on circuits and what operating conditions increase or decrease their effects. [104] is one of the initial works studying the impact of these different aging phenomena. It

showed that EM has a major impact on interconnect operation. Circuit aging due to switching activity was shown in [79], where the aging model is proportional to signal switching probabilities. [129] showed that switching activities impact HCE. [66] shows that HCE affect on SRAM stability while NBTI affects cell transition speed. Further [61] studies the NBTI impact on read stability of SRAM cells.

Detection and diagnosis of permanent faults is vital for continuous operation. Works such as [15] have examined self-calibrating units to identify the onset of aging in circuits while an orthogonal solution to identify aging talks about using the idle time in a circuit to analyze aging using test vectors [100]. This testing can be optimized further by adopting different voltage-frequency levels to identify whether guardbands might get violated [36]. An alternate approach to tackle defects due to aging has been to use invariants in program execution to identify situations when errors occur due to usage of the faulty components [89, 92]. Recovery from hard faults has involved using checkpointing correct system state and recovering using that whenever faults are detected [64]. These works are orthogonal to ours where we try to reduce the impact of aging while these look at symptoms to identify and tolerate faulty components. An alternate approach to tackle detects due to aging has been to use invariants in program execution to identify situations when errors occur due to usage of the faulty components [89, 92]. Besides these, hardware- and software-based techniques have also been proposed to detect defects [20, 63, 32]. Since aging is a slow phenomenon; detection mechanisms cannot place any significant overhead on common-case operation. Recovery from hard faults has involved using checkpointing correct system state and recovering using that whenever faults are detected [64].

Works like [2], [61] propose bit-flipping/inverting techniques to address NBTI in RAM-based structures. These works do not study HCE degradation and the impact of switching activity on HCE and NBTI degradation. For latch-based collapsing queues, restricted collapsing reduces the overall switching activity which has significant impact on the overall degradation. Further our analysis was done on the actual SNM and read degradation using values from actual benchmark runs.

Aging-aware chip-multiprocessor scheduling schemes have been proposed to assign jobs based on tracking aging across cores and by making voltage changes across entire cores [40, 112]. But the granularity of operation is much finer in our

solution.

6.7 Summary

Increasing risk of timing violations in future microprocessor designs has made transistor aging an important problem to address. Two phenomena that have gained importance are NBTI and HCE. In this work we studied their impact on latch-based collapsing issue queues which incorporate cells with significant variation in wearout and hence can detrimentally affect performance over a microprocessor's lifetime. We propose the restricted collapsing infrastructure that which operates on all entries in the issue queue and reduces the worst case aging degradation significantly. The read delay degradation is reduced by as much as 32% for about a 1% loss in performance. Based on architectural requirements, designers could effectively combine this technique with other aging mitigation solutions that address aging in different parts of the pipeline.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

Decreasing transistor sizes combined with the need to meet power and performance requirements have increased reliability concerns in the coming generations of microprocessors. Fault-tolerant microprocessor design is gaining importance with the emergence of newer transistor failure phenomena that prevent the underlying circuitry from operating correctly and from meeting lifetime requirements. The growing reliability concern also means that simpler solutions like redundant entries in array- or queue-based structures, frequency binning to classify chips that operate at different speeds and ECC are no longer effective. These solutions are insufficient to meet the reliability requirements or the associated power, performance and area costs make them unviable to adopt.

Taking these factors into account, this dissertation makes several contributions towards addressing the reliability issues that arise from both transient and permanent faults. We looked at addressing these reliability issues in two important processor structures, the issue queue and reorder buffer. These structures are very critical for high performance within an OOO pipeline. The solutions provided take into account their criticality in determining overall throughput.

Process Variation severely limits the performance benefits of moving to a lower technology generation. We provided mechanisms to mitigate the PV effects in the issue queue. While a PV-unaware solution has about a 21% loss in performance, our design decreases this to a mere 1%. We presented microarchitectural knobs for

controlling the soft error vulnerability of the Reorder Buffer. The design presented can easily be extended to other structures and even cover the entire pipeline. The knobs can be modulated to meet the target reliability budgets specified by system designers. Further the performance overhead, even at high reliability constraints, is only about 8% of a single thread execution. We did a detailed study on the factors affecting the soft error vulnerability of multicores. Soft error vulnerability varies significantly across the individual cores by as much as 50% depending on the application. Varying the total number of cores and application threads showed that while System AVF increases with the number of cores, System Vulnerability need not. Based on these observations we proposed a runtime technique that varied the number of cores on which the application ran. For only a 5% performance overhead, the System Vulnerability was decreased by 37%. Given that soft error vulnerability varies with structure occupancy and operating voltage levels, voltage-frequency scaling has a significant reliability impact. Our analysis showed that the soft error vulnerability varies dramatically with the DVFS algorithm, by as much as 80%, and designers have to aware of the reliability impact of choosing a specific algorithm. Transistor aging limits the lifetime guarantees that designers can provide for their platforms. Solutions to address the aging effects come at the cost of performance. We address NBTI and HCE effects, whose effects are significant at 45nm and lower generations, in the issue queue. For a 1% loss in performance, we reduced the read delay degradation by 39%.

Given multicores with simple in-order cores, where does this work stand? We are at the crossroads of a fundamental shift in architectural design as frequency scaling is getting replaced by core scaling. Hence we are likely to see more cores in future microprocessors and it is likely that each of these cores would be simpler than the current ones. Given such a scenario, has this work addressed and solved a useful problem? Our opinion is it has. Note that while cores are getting simpler, the memory-side of the system is getting more complex due to the need to keep all the cores working to improve overall throughput. As such, the memory controller and other queues related to the multiple levels of caching and prefetching are becoming very similar in design to the ROB and the issue queue and hence are prone to the failure mechanisms we have discussed in this work.

7.2 Future Work

Low power operation is very critical in multicores. As we reach near-threshold operation, the chances of multi-bit errors occuring is increasing in both the caches and core as well. While the solutions proposed in this dissertation to address soft errors are based on the SEU model, modeling the effects of multi-bit errors and providing modular techniques to tolerate these errors will become a definite requirement.

Our PV-aware issue queue, while effective depends on test vectors to identify circuit operating speeds at a fine granularity. Hence there is a greater need to develop variation-aware testing techniques that are cost effective.

Voltage-Frequency scaling also affects the circuit lifetime. Also the increasing impact of PV means that circuits are less robust to operating correctly at different voltages.

In this dissertation, we only looked at the reliability impact with respect to soft errors. Further analysis is needed to study the delay degradation due to operating the system at different voltage levels, especially in multicores.

Bibliography

[1] J. Abella, R. Canal, and A. Gonzalez. Power- and complexity-aware issue queue designs. volume 23, pages 50–58. IEEE Computer Society, 2003.

[2] J. Abella, X. Vera, and A. Gonzalez. Penelope: The nbti-aware processor. *Microarchitecture, 2007. MICRO 2007. 40th Annual IEEE/ACM International Symposium on*, pages 85–96, 1-5 Dec. 2007.

[3] A. Agarwal, B. C. Paul, H. Mahmoodi, A. Datta, and K. Roy. A process-tolerant cache architecture for improved yield in nanoscale technologies. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 13(1):27–38, January 2005.

[4] A. Agarwal, B. C. Paul, S. Mukhopadhyay, and K. Roy. Process variation in embedded memories: failure analysis and variation aware architecture. *Solid-State Circuits, IEEE Journal of*, 40:1804–1814, 2005.

[5] H. Ananthan, C. H. Kim, and K. Roy. Larger-than-vdd forward body bias in sub-0.5v nanoscale cmos. In *ISLPED '04: Proceedings of the 2004 international symposium on Low power electronics and design*, pages 8–13. ACM Press, 2004.

[6] D. K. S. andn Jeff A. Babcock. Negative bias temperature instability: Road to cross in deep submicron silicon semiconductor manufacturing. *Journal of Applied Physics*, 94:1–18, 2003.

[7] D. Arumí-Delgado, R. Rodríguez-Montanés, J. P. de Gyvez, and G. Gronthoud. Process-variability aware delay fault testing of "vt and weak-open defects. In *ETW '03: Proceedings of the 8th IEEE European Test Workshop*, page 85, Washington, DC, USA, 2003. IEEE Computer Society.

[8] Bathtub curve of failure rates. http://parts.jpl.nasa.gov/organization/group-5141/parts-technology/qualification-methodologies/.

[9] G. B. Bell and M. H. Lipasti. Deconstructing commit. In *Proceedings of the 4th International Symposium on Performance Analysis of Systems and Software*, Austin, Texas, March 2004.

[10] M. Bennaser and C. A. Moritz. Power and failure analysis of cam cells due to process variations. In *ICECS '06: 13th IEEE International Conference on Electronics, Circuits and Systems*, pages 608 – 611, 2006.

[11] K. Bernstein, D. J. Frank, A. E. Gattiker, W. Haensch, B. L. Ji, S. R. Nassif, E. J. Nowak, D. J. Pearson, and N. J. Rohrer. High-performance cmos variability in the 65-nm regime and beyond. *IBM J. Res. Dev.*, 50(4/5):433–449, 2006.

[12] K. Bernstein, D. J. Frank, A. E. Gattiker, W. Haensch, B. L. Ji, S. R. Nassif, E. J. Nowak, D. J. Pearson, and N. J. Rohrer. High-performance cmos variability in the 65-nm regime and beyond. *IBM J. Res. Dev.*, 50(4/5):433–449, 2006.

[13] S. Bhardwaj, W. Wang, R. Vattikonda, Y. Cao, and S. Vrudhula. Predictive modeling of the nbti effect for reliable design. *Conference 2006, IEEE Custom Integrated Circuits*, pages 189–192, Sept. 2006.

[14] A. Biswas, P. Racunas, R. Cheveresan, J. S. Emer, S. S. Mukherjee, and R. Rangan. Computing architectural vulnerability factors for address-based structures. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 532–543, 2005.

[15] J. Blome, S. Feng, S. Gupta, and S. Mahlke. Self-calibrating online wearout detection. In *MICRO '07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 109–122, Washington, DC, USA, 2007. IEEE Computer Society.

[16] S. Borkar. Microarchitecture and design challenges for gigascale integration. In *MICRO 37: Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, pages 3–3. IEEE Computer Society, 2004.

[17] S. Borkar. Designing reliable systems from unreliable components: The challenges of transistor variability and degradation. *IEEE Micro*, 25(6):10–16, 2005.

[18] S. Borkar, N. P. Jouppi, and P. Stenstrom. Microprocessors in the era of terascale integration. In *DATE '07: Proceedings of the conference on Design, automation and test in Europe*, pages 237–242, San Jose, CA, USA, 2007. EDA Consortium.

[19] S. Borkar, T. Karnik, S. Narendra, J. Tschanz, A. Keshavarzi, and V. De. Parameter variations and impact on circuits and microarchitecture. In *DAC '03: Proceedings of the 40th conference on Design automation*, pages 338–342. ACM Press, 2003.

[20] F. Bower, D. Sorin, and S. Ozev. A mechanism for online diagnosis of hard faults in microprocessors. In *International Symposium on Microarchitecture*, pages 197–208, 2005.

[21] K. Bowman, S. Duvall, and J. Meindl. Impact of die-to-die and within-die parameter fluctuations on the maximum clock frequency distribution for gigascale integration. *Solid-State Circuits, IEEE Journal of*, 37(2):183–190, February 2002.

[22] E. Brekelbaum, J. Rupley, C. Wilkerson, and B. Black. Hierarchical scheduling windows. In *MICRO 35: Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, pages 27–36. IEEE Computer Society Press, 2002.

[23] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. *SIGARCH Comput. Archit. News*, 28(2):83–94, 2000.

[24] D. Burger and T. Austin. The SimpleScalar Toolset, Version 3.0. http://www.simplescalar.com.

[25] A. Buyuktosunoglu, A. A. El-Moursy, and D. H. Albonesi. An oldest-first selection logic implementation for non-compacting issue queues. In *15th Annual IEEE International ASIC/SOC Conference*, pages 31 – 35, 2002.

[26] C. Constantinescu. Trends and Challenges in VLSI Circuit Reliability. *IEEE Micro*, 23(4):14–19, July-August 2003.

[27] Q. Cai, J. González, R. Rakvic, G. Magklis, P. Chaparro, and A. González. Meeting points: using thread criticality to adapt multicore hardware to parallel regions. In *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 240–249, New York, NY, USA, 2008. ACM.

[28] V. Cakarevic, P. Radojkovic, J. Verdz, A. Pajuelo, R. Gioiosa, F. Cazorla, M. Nemirovsky, and M. Valero. Understanding the overhead of the spinlock loop in cmt architectures. In *In Procs. of Workshop on the Interaction between Operating Systems and Computer Architecture (WIOSCA), held in conjunction with ISCA-35*, June 2008.

[29] V. Chandra and R. C. Aitken. Impact of voltage scaling on nanoscale sram reliability. In *DATE*, pages 387–392, 2009.

[30] M. Chen and A. Orailoglu. Improving circuit robustness with cost-effective soft-error-tolerant sequential elements. In *ATS '07: Proceedings of the 16th Asian Test Symposium*, pages 307–312, Washington, DC, USA, 2007. IEEE Computer Society.

[31] C. Constantinescu. Trends and challenges in vlsi circuit reliability. *IEEE Micro*, 23:14–19, 2003.

[32] K. Constantinides, O. Mutlu, T. Austin, and V. Bertacco. Software–based online detection of hardware defects: Mechanisms, architectural support, and evaluation. In *IEEE Conference on Microarchitecture (MICRO-2007)*, January 2007.

[33] A. Cristal, D. Ortega, J. Llosa, and M. Valero. Out-of-order commit processors. In *HPCA '04: Proceedings of the 10th International Symposium on High Performance Computer Architecture*, page 48, Washington, DC, USA, 2004. IEEE Computer Society.

[34] J. P. S. David E. Culler and A. Gupta. *Parallel Computer Architectures. A Hardware/Software Approach*. Morgan Kaufmann, 1998.

[35] P. Dubey. Recognition, mining and synthesis moves computers to the era of tera. 2005.

[36] D. emerging wearout faults. J. c. smolens, b. t. gold, j. c. hoe, b. falsafi, and k.mai. In *In Proceedings of the IEEE Workshop on Silicon Errors in Logic – System Effects*, 2007.

[37] D. Ernst and T. Austin. Efficient dynamic scheduling through tag elimination. In *ISCA '02: Proceedings of the 29th annual international symposium on Computer architecture*, pages 37–46. IEEE Computer Society, 2002.

[38] D. Ernst, N. S. Kim, S. Das, S. Pant, T. Pham, R. Rao, C. Ziesler, D. Blaauw, T. Austin, and T. Mudge. Razor: A low-power pipeline based on circuit-level timing speculation. In *Micro Conference*, December 2003.

[39] J. Farrell and T. Fischer. Issue logic for a 600-mhz out-of-order execution microprocessor. *Solid-State Circuits, IEEE Journal of*, 33(5), 1998.

[40] S. Feng, S. Gupta, A. Ansari, and S. A. Mahlke. Maestro: Orchestrating lifetime reliability in chip multiprocessors. In Y. N. Patt, P. Foglia, E. Duesterwald, P. Faraboschi, and X. Martorell, editors, *HiPEAC*, volume 5952 of *Lecture Notes in Computer Science*, pages 186–200. Springer, 2010.

[41] X. Fu, T. Li, and J. Fortes. Sim-soda: A framework for microarchitecture reliability analysis. In *Workshop on Modeling, Benchmarking and Simulation (Held in conjunction with ISCA-33)*, 2006.

[42] X. Fu, J. Poe, T. Li, and J. A. B. Fortes. Characterizing microarchitecture soft error vulnerability phase behavior. In *MASCOTS '06: Proceedings of the 14th IEEE International Symposium on Modeling, Analysis, and Simulation*, pages 147–155, Washington, DC, USA, 2006. IEEE Computer Society.

[43] T. Funaki and T. Sato. Formulating mitf for a multicore processor with seu tolerance. In *Digital System Design Architectures, Methods and Tools, 2008. DSD '08. 11th EUROMICRO Conference on*, pages 234–241, Sept. 2008.

[44] K. Ghose. Reducing energy requirements for instruction issue and dispatch in superscalar microprocessors (poster session). In *ISLPED '00: Proceedings of the 2000 international symposium on Low power electronics and design*, pages 231–233. ACM, 2000.

[45] G. Gielen, P. D. Wit, E. Maricau, J. Loeckx, J. Martín-Martínez, B. Kaczer, G. Groeseneken, R. Rodríguez, and M. Nafría. Emerging yield and reliability challenges in nanometer cmos technologies. In *DATE '08: Proceedings of the conference on Design, automation and test in Europe*, pages 1322–1327, New York, NY, USA, 2008. ACM.

[46] M. Gomaa, C. Scarbrough, T. Vijaykumar, and I. Pomeranz. Transient-Fault Recovery for Chip Multiprocessors. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 98–109, June 2003.

[47] M. A. Gomaa and T. N. Vijaykumar. Opportunistic transient-fault detection. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 172–183, 2005.

[48] IDF 2009 - Intel Shows off 22nm and 32nm, Sandy Bridge Demoed. http://www.anandtech.com/show/2842/.

[49] Intel(R). Server consolidation using quad-core processors. 2006.

[50] Intel(R). Intel previews intel xeon(r) 'nehalem-ex' processor. 2009.

[51] A. Iyer and D. Marculescu. Power efficiency of voltage scaling in multiple clock, multiple voltage cores. In *ICCAD '02*, pages 379–386, New York, NY, USA, 2002. ACM.

[52] H. M. Jacobson. Improved clock-gating through transparent pipelining. In *ISLPED '04: Proceedings of the 2004 international symposium on Low power electronics and design*, pages 26–31. ACM Press, 2004.

[53] M. H. J. J. J. M. S. P. C. P. Jeffrey Hicks, Daniel Bergstrom and J. Wiedemer. 45nm transistor reliability. *Intel Technology Journal*, 12(2):131–144, 2008.

[54] R. Kalla, B. Sinharoy, and J. Tendler. Ibm power5 chip: a dual-core multithreaded processor. *Micro, IEEE*, 24(2):40–47, Mar-Apr 2004.

[55] V. Kazempour, A. Fedorova, and P. Alagheband. Performance implications of cache affinity on multicore processors. In *Euro-Par '08: Proceedings of the 14th international Euro-Par conference on Parallel Processing*, pages 151–161, Berlin, Heidelberg, 2008. Springer-Verlag.

[56] I. Kim and M. H. Lipasti. Half-price architecture. In *ISCA '03: Proceedings of the 30th annual international symposium on Computer architecture*, pages 28–38. ACM Press, 2003.

[57] I. Kim and M. H. Lipasti. Understanding scheduling replay schemes. In *HPCA '04: Proceedings of the 10th International Symposium on High Performance Computer Architecture*, page 198, Washington, DC, USA, 2004. IEEE Computer Society.

[58] K. Krewell. Ultrasparc iv mirrors predecessor: Sun builds dual-core chip in 130nm. In *Microprocessor Report*, Nov 2003.

[59] G. Kucuk, D. Ponomarev, and K. Ghose. Low-complexity reorder buffer architecture. In *ICS '02: Proceedings of the 16th International Conference on Supercomputing*, pages 57–66. ACM Press, 2002.

[60] H. Kufluoglu. *MOSFET Degradation due to Negative Bias Temperature Instability (NBTI) and Hot Carrier Injection (HCI) and its implications for reliability-aware VLSI design*. PhD thesis, Purdue University, 2007.

[61] S. V. Kumar, C. H. Kim, and S. S. Sapatnekar. Impact of nbti on sram read stability and design for reliability. In *ISQED '06: Proceedings of the 7th International Symposium on Quality Electronic Design*, pages 210–218, Washington, DC, USA, 2006. IEEE Computer Society.

[62] H. Li, S. Bhunia, Y. Chen, T. N. Vijaykumar, and K. Roy. Deterministic clock gating for microprocessor power reduction. In *HPCA '03: Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, page 113, Washington, DC, USA, 2003. IEEE Computer Society.

[63] M.-L. Li, P. Ramachandran, S. Sahoo, S. Adve, V. Adve, and Y. Zhou. Swat: An error resilient system. In *the Fourth Workshop on Silicon Errors in Logic - System Effects (SELSE - IV)*, March 2008.

[64] M.-L. Li, P. Ramachandran, S. Sahoo, S. Adve, V. Adve, and Y. Zhou. Trace-based microarchitecture-level diagnosis of permanent hardware faults. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, June 2008.

[65] X. Li, S. V. Adve, P. Bose, and J. A. Rivers. Softarch: An architecture level tool for modeling and analyzing soft errors. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pages 496–505, 2005.

[66] X. Li, J. Qin, B. Huang, X. Zhang, and J. Bernstein. Sram circuit-failure modeling and reliability simulation with spice. *Device and Materials Reliability, IEEE Transactions on*, 6(2):235–246, June 2006.

[67] X. Liang and D. Brooks. Mitigating the impact of process variations on processor register files and execution units. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 504–514. IEEE Computer Society, 2006.

[68] S.-Z. E. Lin, C. Changfan, Y.-C. Hsu, and F.-S. Tsai. Optimal time borrowing analysis and timing budgeting optimization for latch-based designs. *ACM Trans. Des. Autom. Electron. Syst.*, 7(1):217–230, 2002.

[69] G. Magklis, P. Chaparro, J. González, and A. González. Independent front-end and back-end dynamic voltage scaling for a gals microarchitecture. In *ISLPED '06*, pages 49–54, New York, NY, USA, 2006. ACM.

[70] D. Marculescu and E. Talpes. Variability and energy awareness: a microarchitecture-level perspective. In *DAC '05: Proceedings of the 42nd annual conference on Design automation*, pages 11–16. ACM Press, 2005.

[71] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's general execution-driven multiprocessor simulator (gems) toolset. *SIGARCH Comput. Archit. News*, 33(4):92–99, 2005.

[72] J. W. McPherson. Reliability challenges for 45nm and beyond. In *DAC '06: Proceedings of the 43rd annual Design Automation Conference*, pages 176–181, New York, NY, USA, 2006. ACM.

[73] S. Mukherjee. *Architecture Design for Soft Errors.* Morgan Kaufmann, 2008.

[74] S. Mukherjee, M. Kontz, and S. Reinhardt. Detailed Design and Evaluation of Redundant Multithreading Alternatives. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 99–110, May 2002.

[75] S. Mukherjee, C. Weaver, J. Emer, S. Reinhardt, and T. Austin. A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 29–40, December 2003.

[76] A. Mupid, M. Mutyam, N. Vijaykrishnan, Y. Xie, and M. J. Irwin. Variation analysis of cam cells. In *ISQED '07: 8th International Symposium on Quality of Electronic Design*, pages 333 – 338. IEEE Computer Society, 2007.

[77] S. Murali, A. Mutapcic, D. Atienza, R. Gupta, S. Boyd, L. Benini, and G. De Micheli. Temperature control of high-performance multi-core platforms using convex optimization. In *DATE '08: Proceedings of the conference on Design, automation and test in Europe*, pages 110–115, New York, NY, USA, 2008. ACM.

[78] E. Musoll. A thermal-friendly load-balancing technique for multi-core processors. In *ISQED '08: Proceedings of the 9th International Symposium on Quality Electronic Design (isqed 2008)*, pages 549–552, Washington, DC, USA, 2008. IEEE Computer Society.

[79] F. N. Najm. Transition density, a stochastic measure of activity in digital circuits. In *DAC '91: Proceedings of the 28th conference on ACM/IEEE design automation*, pages 644–649, New York, NY, USA, 1991. ACM.

[80] S. Palacharla. *Complexity-Effective Superscalar Processors*. PhD thesis, University of Wisconsin - Madison, 1998.

[81] S. Palacharla, N. Jouppi, and J. Smith. Complexity-Effective Superscalar Processors. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 206–218, June 1997.

[82] A. Parashar, S. Gurumurthi, and A. Sivasubramaniam. A Complexity-Effective Approach to ALU Bandwidth Enhancement for Instruction-Level Temporal Redundancy. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 376–386, June 2004.

[83] A. Parashar, S. Gurumurthi, and A. Sivasubramaniam. Slick: Slice-based locality exploitation for efficient redundant multithreading. In *Proceedings of the Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 2006.

[84] Public Roadmap Desktop, Mobile and Data Center - 2010. http://download.intel.com/products/roadmap/roadmap.pdf.

[85] K. Puttaswamy and G. H. Loh. Thermal herding: Microarchitecture techniques for controlling hotspots in high-performance 3d-integrated processors. In *HPCA '07: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 193–204, Washington, DC, USA, 2007. IEEE Computer Society.

[86] K. Raghavendra and M. Mutyam. Process variation aware issue queue design. In *DATE '08: Proceedings of the conference on Design, automation and test in Europe*, 2008.

[87] M. Rashid and M. Huang. Supporting highly-decoupled thread-level redundancy for parallel programs. In *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on*, pages 393–404, Feb. 2008.

[88] J. Ray, J. Hoe, and B. Falsafi. Dual Use of Superscalar Datapath for Transient-Fault Detection and Recovery. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 214–224, December 2001.

[89] V. Reddv, A. Al-Zawawi, and E. Rotenberg. Assertion-based microarchitecture design for improved fault tolerance. *Computer Design, 2006. ICCD 2006. International Conference on*, pages 362–369, 1-4 Oct. 2007.

[90] V. K. Reddy, S. Parthasarathy, and E. Rotenberg. Understanding prediction-based partial redundant threading for low-overhead, high-coverage fault tolerance. In *Proceedings of the Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 2006.

[91] S. Reinhardt and S. Mukherjee. Transient Fault Detection via Simultaneous Multithreading. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 25–36, June 2000.

[92] S. Sahoo, M.-L. Li, P. Ramachandran, S. Adve, V. Adve, and Y. Zhou. Using likely program invariants to detect hardware errors. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, June 2008.

[93] B. Schroeder and G. A. Gibson. Disk failures in the real world: what does an mttf of 1,000,000 hours mean to you? In *FAST '07: Proceedings of the 5th USENIX conference on File and Storage Technologies*, page 1, Berkeley, CA, USA, 2007. USENIX Association.

[94] G. Semeraro, D. H. Albonesi, S. G. Dropsho, G. Magklis, S. Dwarkadas, and M. L. Scott. Dynamic frequency and voltage control for a multiple clock

domain microarchitecture. In *MICRO 35: Proceedings of the 35th annual ACM/IEEE international symposium on Microarchitecture*, pages 356–367. IEEE Computer Society Press, 2002.

[95] G. Semeraro, D. H. Albonesi, S. G. Dropsho, G. Magklis, S. Dwarkadas, and M. L. Scott. Dynamic frequency and voltage control for a multiple clock domain microarchitecture. In *MICRO 35*, pages 356–367, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[96] Semiconductor International. http://www.lrcx.org/downloads/$Technical_{P}ubs/SI_{2}008-$ $01_{3}2nm_{M}arked_{b}y_{L}itho$.pdf.

[97] J. Shen and M. Lipasti. *Modern Processor Design: Fundamentals of Superscalar Processors (Beta Edition)*. McGraw Hill, 2003.

[98] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically Characterizing Large Scale Program Behavior. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 2002.

[99] P. Shivakumar, M. Kistler, S. Keckler, D. Burger, and L. Alvisi. Modeling the Effect of Technology Trends on Soft Error Rate of Combinational Logic. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, June 2002.

[100] S. Shyam, K. Constantinides, S. Phadke, V. Bertacco, and T. Austin. Ultra low-cost defect protection for microprocessor pipelines. *SIGPLAN Not.*, 41(11):73–82, 2006.

[101] K. Skadron, M. R. Stan, K. Sankaranarayanan, W. Huang, S. Velusamy, and D. Tarjan. Temperature-aware microarchitecture: Modeling and implementation. *ACM Trans. Archit. Code Optim.*, 1(1):94–125, 2004.

[102] J. C. Smolens, B. T. Gold, B. Falsafi, and J. C. Hoe. Reunion: Complexity-effective multicore redundancy. In *Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International Symposium on*, pages 223–234, Dec. 2006.

[103] N. Soundararajan, N. Vijaykrishnan, and A. Sivasubramaniam. Impact of dynamic voltage and frequency scaling on the architectural vulnerability of gals architectures. In *ISLPED '08: Proceeding of the thirteenth international symposium on Low power electronics and design*, pages 351–356, New York, NY, USA, 2008. ACM.

[104] J. Srinivasan, S. Adve, P. Bose, and J. Rivers. The impact of technology scaling on lifetime reliability. In *International Conference on Dependable Systems and Networks*, pages 177–186, 2004.

[105] S. Srinivasan, R. Krishnan, P. Mangalagiri, Y. Xie, V. Narayanan, M. J. Irwin, and K. Sarpatwari. Toward increasing fpga lifetime. *IEEE Trans. Dependable Secur. Comput.*, 5(2):115–127, 2008.

[106] K. Sundaramoorthy, Z. Purser, and E. Rotenburg. Slipstream processors: improving both performance and fault tolerance. In *ASPLOS-IX: Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, pages 257–268, 2000.

[107] D. Sylvester, D. Blaauw, and E. Karl. Elastic: An adaptive self-healing architecture for unpredictable silicon. *IEEE Des. Test*, 23(6):484–490, 2006.

[108] E. Talpes and D. Marculescu. A critical analysis of application-adaptive multiple clock processors. In *ISLPED '03*, pages 278–281, New York, NY, USA, 2003. ACM.

[109] E. Talpes and D. Marculescu. Toward a multiple clock/voltage island design style for power-aware processors. *IEEE Trans. Very Large Scale Integr. Syst.*, 13(5):591–603, 2005.

[110] M. Tehranipour, Z. Navabi, and S. Falkhrai. An efficient bist method for testing of embedded srams. In *Proceedings of IEEE International Symposium on Circuits and Systems*, 2001.

[111] A. Tiwari, S. R. Sarangi, and J. Torrellas. Recycle: Pipeline adaptation to tolerate process variation. In *ISCA '07: Proceedings of the 30th annual international symposium on Computer architecture*, 2007.

[112] A. Tiwari and J. Torrellas. Facelift: Hiding and slowing down aging in multicores. In *MICRO '08: Proceedings of the 2008 41st IEEE/ACM International Symposium on Microarchitecture*, pages 129–140, Washington, DC, USA, 2008. IEEE Computer Society.

[113] J. Tschanz, K. Bowman, and V. De. Variation-tolerant circuits: circuit solutions and techniques. In *DAC '05: Proceedings of the 42nd annual Design Automation Conference*, pages 762–763, New York, NY, USA, 2005. ACM.

[114] D. Tullsen, S. Eggers, and H. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 392–403, June 1995.

[115] K. Ünlü, V. Narayanan, S. M.Cetiner, V. Degalahal, and M. J. Irwin. Neutron-induced soft error rate measurements in semiconductor memories. *Nuclear Instruments and Methods in Physics Research A*, 579:252–255, 2007.

[116] O. S. Unsal, J. W. Tschanz, K. Bowman, V. De, X. Vera, A. Gonzalez, and O. Ergin. Impact of parameter variations on circuits and microarchitecture. *IEEE Micro*, 26(6):30–39, 2006.

[117] T. Vijaykumar, I. Pomeranz, and K. Cheng. Transient-Fault Recovery via Simultaneous Multithreading. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 87–98, May 2002.

[118] K. R. Walcott, G. Humphreys, and S. Gurumurthi. Dynamic prediction of architectural vulnerability from microarchitectural state. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, pages 516–527, New York, NY, USA, 2007. ACM.

[119] N. J. Wang and S. J. Patel. Restore: Symptom based soft error detection in microprocessors. In *Proceedings of the 2005 International Conference on Dependable Systems and Networks (DSN'05)*, pages 30–39, 2005.

[120] W. Wang, V. Reddy, A. Krishnan, R. Vattikonda, S. Krishnan, and Y. Cao. Compact modeling and simulation of circuit reliability for 65nm cmos technology. *IEEE Transactions on Devices and Materials Reliability*, 7(4):509–517, Dec. 2007.

[121] W. Wang, S. Yang, S. Bhardwaj, R. Vattikonda, S. Vrudhula, F. Liu, and Y. Cao. The impact of nbti on the performance of combinational and sequential circuits. In *DAC '07: Proceedings of the 44th annual conference on Design automation*, pages 364–369, New York, NY, USA, 2007. ACM.

[122] C. Weaver, J. Emer, S. Mukherjee, and S. Reinhardt. Techniques to Reduce the Soft Error Rate of High-Performance Microprocessor. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 264–275, June 2004.

[123] P. Wells and G. Sohi. Serializing instructions in system-intensive workloads: Amdahl's law strikes again. In *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on*, pages 264–275, Feb. 2008.

[124] P. M. Wells, K. Chakraborty, and G. S. Sohi. Hardware support for spin management in overcommitted virtual machines. In *PACT '06: Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, pages 124–133, New York, NY, USA, 2006. ACM.

[125] P. M. Wells, K. Chakraborty, and G. S. Sohi. Mixed-mode multicore reliability. In *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, pages 169–180, New York, NY, USA, 2009. ACM.

[126] T. F. Wenisch, S. Somogyi, N. Hardavellas, J. Kim, A. Ailamaki, and B. Falsafi. Temporal streaming of shared memory. *SIGARCH Comput. Archit. News*, 33(2):222–233, 2005.

[127] World's First 2-Billion Transistor Microprocessor. http://www.intel.com/technology/architecture-silicon/2billion.htm.

[128] Q. Wu, P. Juang, M. Martonosi, and D. W. Clark. Formal online methods for voltage/frequency control in multiple clock domain microprocessors. *SIGARCH Comput. Archit. News*, 32(5):248–259, 2004.

[129] X. Xuan, A. Chatterjee, and A. Singh. Local redesign for reliability of cmos digital circuits under device degradation. *Reliability Physics Symposium Proceedings, 2004. 42nd Annual. 2004 IEEE International*, pages 651–652, 25-29 April 2004.

[130] W. Zhang, X. Fu, T. Li, and J. Fortes. An analysis of microarchitecture vulnerability to soft errors on simultaneous multithreaded architectures. In *Performance Analysis of Systems and Software, 2007. ISPASS 2007. IEEE International Symposium on*, pages 169–178, April 2007.

[131] W. Zhang and T. Li. Managing multi-core soft-error reliability through utility-driven cross domain optimization. In *ASAP '08: Proceedings of the 2008 International Conference on Application-Specific Systems, Architectures and Processors*, pages 132–137, Washington, DC, USA, 2008. IEEE Computer Society.

[132] W. Zhao and Y. Cao. New generation of predictive technology model for sub-45nm design exploration. In *ISQED '06: Proceedings of the 7th International Symposium on Quality Electronic Design*, pages 585–590. IEEE Computer Society, 2006.

[133] Y. Zhu, D. H. Albonesi, and A. Buyuktosunoglu. A high performance, energy efficient gals processor microarchitecture with reduced implementation complexity. In *ISPASS '05*, pages 42–53, Washington, DC, USA, 2005. IEEE Computer Society.

Vita

Niranjan Soundararajan

Niranjan Soundararajan is from the cosmopolitan city of Chennai in southern India. He obtained his Bachelors in Computer Science and Engineering from Sri Venkateswara College of Engineering, University of Madras, in the year 2004. He joined the doctoral program in the Department of Computer Science and Engineering at Pennsylvania State University in 2005. During his doctoral program, he published in reputed conferences in his research area that include ISCA, DSN, ISLPED and ACM SIGMETRICS besides co-authoring with other people in his lab. He has reviewed papers for several IEEE and ACM conferences and journals including ISCA, HPCA, Micro, PACT, ICCAD, DAC, TCAD and TVLSI. He interned at Intel for 6 months and AMD for 3 months. He is a student member of ACM.