

ABSTRACT

Title of Thesis: Reducing the Soft Error Rates
of a High-Performance Microprocessor
Using Front-End Throttling

Smitha M Kalappurakkal, Master of Science, 2006

Thesis directed by: Professor Manoj Franklin
Department of Electrical & Computer Engineering

Microprocessors are increasingly used in a variety of applications from small handheld calculators to multi-million dollar servers. With our increasing dependence on microprocessor-based systems, greater importance needs to be given not only to a processor's performance but also to its dependability. With each new technology generation, we witness a consistent increase in cosmically induced soft errors per chip. Earlier, only memory structures were affected significantly by soft errors. But today, most memories are well protected by error detection and correction codes. However, unprotected logic state elements, which were not a great concern in older technology generations, are increasingly becoming a concern due to the technology scaling trend. Although the fault rate per transistor has been remaining roughly the same across generations, the increasing number of transistors per chip is resulting in a steady increase in the raw error rates. Thus, the increased functionality and performance as dictated by Moore's Law comes at the cost of an exponentially increasing soft error rate.

In this thesis, we present techniques to reduce a processor’s soft error rate. We focus on one of the major contributors of the on-chip soft error rates - the Instruction Issue Queue(IQ), which is proven to have a significantly higher vulnerability factor (32.7% as measured by our work) compared to other microarchitectural structures like the register file (18.65%), re-order buffer (28%) and execution units (9%). Modern processors often aggressively fetch and decode instructions, in order to exploit as much parallelism as possible. However, this often results in instructions being fetched much earlier than necessary, causing valid instructions to reside in the vulnerable IQ for many needless cycles, waiting for dependencies to be resolved or to be squashed on a mis-speculation event. Additional ILP that may get exposed as a result of this aggressive front-end design, often does not result in any major performance benefits. We exploit this inefficiency by slowing down the front-end of the pipeline when it is not likely to affect the performance significantly. For this, we explore a set of reliability-aware front-end throttling schemes, to bring down the utilization of the IQ, and hence the soft error rates.

We observe the improvement in the structure’s Architectural Vulnerability Factor(AVF), which is defined as the probability that a fault in the structure will result in an externally visible error, on employing the throttling techniques. Our simulations show the AVF of a microprocessor’s instruction queue to be 32.7%. Upon introducing front end throttling schemes, we observe a reduction in the AVF and hence the soft error rates by up to 26.3% with an average performance degradation of less than 2.6%.

Reducing the Soft Error Rates
of a High-Performance Microprocessor
Using Front-End Throttling

by

Smitha M Kalappurakkal

Thesis submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Master of Science
2006

Advisory Committee:
Professor Manoj Franklin, Chair/Advisor
Professor Peter Petrov
Professor Gang Qu

© Copyright by
Smitha M Kalappurakkal
2006

DEDICATION

I dedicate this thesis to my family. Thank you for being there for me, and believing in me, always.

ACKNOWLEDGMENTS

First and foremost, I would like to thank my advisor, Professor Manoj Franklin, for his continued and valuable guidance throughout my course of study at the University of Maryland. He has always been very understanding and supportive.

I would like to extend my gratitude to Professor Gang Qu and Professor Peter Petrov for agreeing to be part of my Thesis committee.

I thank my husband for his constant support and encouragement. I am indebted to my parents and my sister for their advice and prayers for me. Thanks to all my family for inspiring me to do my best.

TABLE OF CONTENTS

List of Tables	vi
List of Figures	vii
1 INTRODUCTION	1
1.1 Soft Errors	1
1.2 Addressing Soft Errors	3
1.3 Our Contribution	4
1.4 Thesis Outline	5
2 SOFT ERRORS: BACKGROUND AND TERMINOLOGY	6
2.1 SDC and DUE Errors	6
2.2 Architectural Vulnerability Factor	8
2.3 Computing SDC Rates	9
2.4 Estimating Reliability-Performance Trade-off	10
3 MOTIVATION AND RELATED WORK	12
3.1 Motivation	12
3.2 Related Work	15
4 ENHANCING RELIABILITY	20
4.1 AVF Computation	20
4.1.1 ACE Analysis	21
4.1.2 Computing AVF using a Performance Model	24
4.2 Reliability-aware Front-end Throttling	26
4.2.1 Occupancy based Throttling	28
4.2.2 Flow based Throttling	30
4.2.3 Rate based Throttling	31
4.3 Overhead Incurred	33
5 EXPERIMENTAL RESULTS	34
5.1 Simulation Model	34
5.2 AVF of the Instruction Queue	36
5.3 Effects of Front-End Throttling	38
5.3.1 Reduction in the Residency Cycles in IQ before Issue	38
5.3.2 Improvement in AVF	38
5.3.3 Degradation in Performance	39
5.3.4 Reliability-Performance Trade-off	41
5.4 Observations	42
6 SUMMARY AND CONCLUSION	44
6.1 Summary	44
6.2 Conclusion	45

LIST OF TABLES

5.1	Processor Configuration	35
5.2	SPEC2000 Benchmarks used and Instructions skipped.	36

LIST OF FIGURES

2.1	Possible Outcomes due to a Faulty Bit in a Microprocessor	7
3.1	Impact of AVF on SDC Rates	13
4.1	Pipeline with Control Logic for Occupancy Based Throttling	29
4.2	Pipeline with Control Logic for Flow Based Throttling	30
4.3	Pipeline with Control Logic for Rate Based Throttling	32
5.1	Bit-level Classification of the IQ Contents	37
5.2	Effect on the Average Number of Residence Cycles in IQ before Issue	39
5.3	Improvement in AVF	40
5.4	Degradation in Performance	41
5.5	Improvement in MITF	42

Chapter 1

INTRODUCTION

Performance has been the driving force of modern microprocessor design for several decades. Over the past several technology generations, we have witnessed an exponential increase in the transistor count per chip as per Moore's law and correspondingly a tremendous progress in the performance and functionality of semiconductor devices in general, and microprocessors in particular. However, at present, this fast-paced progress is challenged by a multitude of factors, among which radiation induced soft errors is a major one.

1.1 Soft Errors

Even if a microprocessor is shipped without any manufacturing defects or hardware design errors, environmental conditions can result in temporary hardware failures. These failures, called soft errors (or transient errors, or single event upsets (SEUs)) are caused by radiation, mainly from neutron particles due to cosmic rays that come from deep outer space [1]. Other causes for soft errors are alpha particles, which arise from naturally occurring radioactive sources, such as contaminants in the chip packaging itself. These energetic radiations, while passing through a semiconductor device, generate electron-hole pairs, which in turn cause charge accumulation at the transistor source and diffusion nodes. Sufficient amounts of

accumulated charge can potentially flip the state of a logic device such as an SRAM cell or a latch. Thus they result in transient, non-repetitive errors in data that are unrelated to components or manufacturing failures. These types of errors are termed as transient, as they do not result in a permanent failure of the victim device.

Among the various device circuit parameters that influence the soft error rate are, the quantity of charge stored and the vulnerable cross-section area. As feature size diminishes, the amount of charge per device also reduces, making it more likely for a particle strike to cause an error. But on the other hand, the reduced cross-section area makes a strike on any given device less likely. Hence, the soft error rate of an individual transistor is expected to remain more or less constant, across technology generations in the near future [3], [1]. However, the per chip soft error rates would increase steadily, in direct proportion to the number of transistors per chip, unless we add better error detection/correction schemes or use more robust technology such as partially depleted SOI. Thus, the exponential increase in transistor count and performance given by Moore's law comes at the expense of exponential increases in error rates for unprotected chips.

As soft errors can cause systems to malfunction with complete unpredictability, extensive techniques are employed to ensure high soft error tolerance for safety critical applications such as aerospace and nuclear systems. But, employing similar techniques for commercial processors is not practical. Physical solutions such as shielding within the IC packaging are hard, since there are no feasible practical absorbents for the neutron particles. In short, cost and performance cannot be sacrificed beyond a point, while the level of fault tolerance should be high enough.

In most cases, soft errors can draw the line between commercial success and failures. Hence, soft errors are now considered as a serious issue by most commercial processor manufacturers.

1.2 Addressing Soft Errors

Intuitively, memory elements would be most susceptible to soft error failures, because of the smaller transistor sizes preferred for them. But, today, most memories are well protected by error detection and correction codes. However, unprotected logic state elements, which were not a great concern in older technology generations due to their bigger sizes, are also increasingly becoming a concern due to the technology scaling trend.

The most obvious method to address the soft error problem is to add more error correction and recovery mechanisms, to the design. Unfortunately, this has side effects in the form of significant degradation of performance, and increased power consumption. Logic level redundancy techniques such as Dual Interlocked Cell [15] can approximately double the number of transistors needed per device. Traditional error correction schemes employed for memory elements have significant storage and time overheads. Redundancy-based schemes (e.g., [17], [18]) would be an over-design for most of today's commercial processors, since these processors can tolerate some level of soft-errors and do not really require bullet-proof operation [19].

1.3 Our Contribution

In a modern superscalar processor, the front-end instruction delivery forms a significant part of the pipeline. Instructions are fetched in program order and dispatched into an issue queue (IQ), from where they are issued into functional units in an out of order fashion. In our work, we propose methods to reduce the soft error vulnerability of an instruction issue queue. Our effort to address the soft errors in microprocessors can be divided into two steps. First, we developed a method to measure and accurately quantify the soft error vulnerability of the IQ. Secondly, we explored methods to bring down the soft error rate.

Researchers have come up with various techniques to diminish or even eliminate the soft errors, but they all come at a price - substantial degradation in cost and performance. In our work, we propose a novel low-cost method for dependability enhancement by bringing down the soft error vulnerability of the instruction issue queue of a modern microprocessor. We show that the performance and hardware overhead for this is negligible, while the improvement in the soft error rates is substantial. We present the improvement observed on the soft error sensitivity of the IQ on employing our method. We also estimated the relative trade-offs between performance and reliability achieved by measuring the MITF (mean instructions to failure). The hardware and performance overhead caused, if any, are quantified.

1.4 Thesis Outline

The rest of this thesis is organized as follows. Chapter 2 describes the background and terminology related to soft errors. Chapter 3 discusses the motivation behind our work, and related prior work. Chapter 4 explains the methodology used to quantify the AVF of the IQ and discusses simple schemes to bring down the same. In chapter 5, we discuss our experimental results and Chapter 6 summarizes and concludes our work.

Chapter 2

SOFT ERRORS: BACKGROUND AND TERMINOLOGY

This chapter describes the background and the soft error terminology which we have followed throughout the remainder of this thesis. Also explained are the metrics used to quantify the soft error rates.

2.1 SDC and DUE Errors

Figure 2.1 [2] illustrates the outcomes that are possible due to a faulty bit. As the figure shows, several of the possible outcomes do not result in a user-visible error in the final program output. When the bit has no error protection, and this bit influences the program outcome as well, an error in that bit results in a *Silent Data Corruption* or *SDC*. This is the most deleterious form of error, since it results in a visible error and goes undetected. To avoid SDC, error detection mechanisms such as parity can be employed, which can detect an error, but not correct it. When the faulty bit has only error protection, and not correction, this is said to result in a *Detected Unrecoverable Error* or *DUE*. Here, a possible error happening at the output can be avoided, by stopping the application on detecting an error. DUE errors can be further classified into *false DUE* and *true DUE* errors. As the names imply, true DUE errors affect the final outcome of the execution while false DUE errors are benign errors. The detection-only scheme does not reduce the error rate, but ensures that the output is not corrupted. In our work, we explore methods to

reduce the SDC rates.

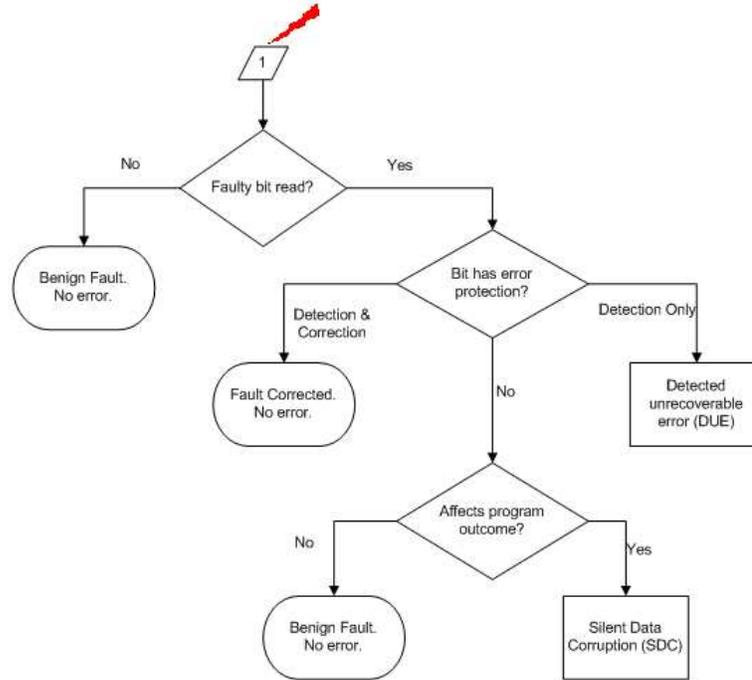


Figure 2.1: Possible Outcomes due to a Faulty Bit in a Microprocessor

Currently the industry specifies soft error rates in terms of SDC and DUE numbers, and these are typically expressed using different metrics by vendors. Traditionally, MTTF (Mean Time to Failure) is used as the appropriate metric to measure system reliability. Other units commonly used are FIT and MTBF. MTBF (Mean time between failures) is based on the interval between failures. MTBF of a component, as the name suggests, is the average time between failures. FIT (Failure in Time) is inversely related to MTBF. One FIT specifies one error in one billion operating hours. A zero error rate implies infinite MTBF and zero FIT. The overall FIT rate of a chip is calculated as the sum of the effective FIT rates of all the structures on-chip. Currently, typical FIT rate numbers for latches and SRAM cells

vary from 0.001 FIT/bit to 0.01 FIT/bit at sea level and are projected to remain the same in the next several technology generations [1], [3], [5], [6].

2.2 Architectural Vulnerability Factor

Not all faults in a microarchitectural structure would affect a program's final outcome. Hence, we cannot come up with an error rate estimate based only on raw device fault rates. Such a pessimistic estimate might lead to over-design of the processor's fault handling features. The net fault rate per bit depends on the device's *vulnerability factors*.

A structure's *Architectural Vulnerability Factor (AVF)* or soft error sensitivity factor is formally defined as the probability that a fault in that processor structure will result in a user visible error in the final program output [6]. In our work, we explore the SDC AVF, specifically for the Instruction Issue Queue. The reader should interpret any reference to AVF as the SDC AVF. In order to estimate the AVF of a structure, we need to figure out which bits in the structure affect the final outcome of the program and which bits doesn't matter.

Mukherjee et al introduced a classification of all the bits in a structure into two groups: *ACE bits* and *un-ACE bits* [6]. ACE bits are those bits (in the structure) that are required for *Architecturally Correct Execution*. On the other hand, un-ACE bits are those bits that are *un-necessary for Architecturally Correct Execution*. For example, a *valid bit* in the IQ, which distinguishes between an empty entry and a valid instruction entry in the IQ, will always be ACE. If a bit flip

happens in this bit, it will almost certainly affect the program execution. On the other hand, the bits in a speculatively loaded wrong path instruction entry would be classified un-ACE, since these will not affect the program outcome at all. Hence, if the bit flip happens to an un-ACE bit, the program outcome will be un-affected. The AVF of a structure at any time can thus be expressed as:

$$\text{AVF} = \frac{\text{number of ACE bits in the structure}}{\text{total number of bits in the structure}}$$

2.3 Computing SDC Rates

This section explains how a microprocessor’s SDC rate can be computed, given the raw error rate of the underlying circuit technology. Vendors usually specify targets for the error rates of a processor. For example, IBM targets 1000 years MTBF for SDC errors. Raw error rates as well as the SDC rates are typically expressed in FIT. In our work, we assume a *Single Event Upset* fault model. That is, we consider only errors caused by single-bit flips. Multi-bit faults are caused by single/multiple particle strikes which affect multiple bits in the structure. They can potentially cause SDC events in structures with single bit error detection like parity. We assume that the probability of multi-bit faults is negligible in comparison to that of single bit faults.

The SDC rate of a processor is the sum of the SDC rate contributions from all its devices. The SDC rate of a device, in turn, is the product of its raw error rate and SDC AVF. A device that has any form of error detection or correction will have zero

SDC AVF and hence zero SDC rate. The SDC AVF of unprotected devices varies according to their nature. For example, the Committed Program Counter will have 100% AVF. This is because any fault in the Program Counter will certainly cause a wrong instruction to be executed, changing the program execution path itself. Whereas, if you consider a branch predictor, even if a fault happens in the predictor structures, the only effect that this might have is that the branch predictions will be wrong. This error will be detected at a stage further down the pipeline, up on which, the necessary remedial action will be taken. The overall program outcome remains unaffected and hence, the AVF of the branch predictor structure is 0%. However, the AVF of other structures on-chip like the Instruction Queues and Register Files, are not as directly intuitive. The AVF figures for these complex structures fall in between 0% and 100%. As mentioned previously, the fault rate of a structure is the product of its raw error rate and its AVF. And, the overall processor fault rate is obtained by summing up the contributions by its individual devices. The knowledge of the effective error rates of a processor will help a designer to achieve the processor's target error rates in a cost effective manner. Further, a knowledge of the error rates of the individual microarchitectural structures is essential to prioritize the fault tolerance features.

2.4 Estimating Reliability-Performance Trade-off

Traditionally, MTTF (Mean Time to Failure) is used as the appropriate metric to measure system reliability. It is defined as:

$$\text{MTTF} = \frac{1}{\text{raw error rate} \times \text{AVF}}$$

But, this metric fails to accurately capture the relative trade-offs between performance and reliability. For example, if there are two systems where the first system is twice as fast as the second but half as reliable, then the probability of a fault occurring in both systems, during the execution of an application, is the same. In this perspective, both systems are equally reliable. However, the MTTF values would suggest that the second, slower system is more reliable, which is not really the case. In order to address this, Weaver et al. [19] introduced a new metric called the *Mean Instructions To Failure (MITF)*. This metric effectively captures the trade-off between performance and reliability.

As the name implies, MITF gives the number of instructions that a processor will commit, on average, between two errors. In [19], Weaver et al defines MITF as:

$$\begin{aligned} \text{MITF} &= \frac{\text{number of committed instructions}}{\text{number of errors encountered}} \\ &= \frac{\text{number of committed instructions}}{\frac{\text{total execution time in cycles}}{\text{frequency} \times \text{MTTF}}} \\ &= \text{IPC} \times \text{frequency} \times \text{MTTF} \\ &= \frac{\text{IPC} \times \text{frequency}}{\text{raw error rate} \times \text{AVF}} \\ &= \frac{\text{frequency}}{\text{raw error rate}} \times \frac{\text{IPC}}{\text{AVF}} \end{aligned}$$

Assuming a constant raw error rate and frequency, MITF is directly proportional to $\frac{\text{IPC}}{\text{AVF}}$. Hence, any method which reduces both AVF and IPC, can be proven as worthwhile only if it also improves the MITF, so that the performance degradation (if any) can be proven lesser in comparison with the decrease in AVF.

Chapter 3

MOTIVATION AND RELATED WORK

3.1 Motivation

Soft errors due to cosmic rays have already made an impact in industry. There are a handful of documented events in industry to substantiate this. In the fifth generation SPARC64 processor, Fujitsu have protected 80% of the total 200k latches by parity check error detection, to counter cosmic ray strikes. The multiply/divide units are protected with residue check and parity prediction circuits [8]. Normand, E [5] has published several incidents of cosmic ray strikes in 1996. Sun Microsystems had acknowledged, in year 2000, that cosmic ray strikes on unprotected L2 cache memories on UltraSPARC IIs caused its Ultra Enterprise servers to crash randomly at several customer sites (AOL, eBay, Verisign and dozens of other corporations were affected) [16]. Sun's UltraSPARC T1 processor has its integer and floating point register files protected by ECC, an extensive level of protection matched only by mainframe-class processors [24].

AVF impacts the extent of error protection required in a microprocessor. Vendors typically specify system error-rate targets at a reference altitude. For example, for its Power4 processor-based systems, IBM's system SDC error-rate target is 1000 years MTBF (equivalent to 114 FIT). Figure 3.1 illustrates why improving the AVF of a structure is important and how it impacts the error protection features of a pro-

cessor. In the figure, we assume 400000 bits to be vulnerable to cosmic ray strikes, and an FIT/bit rate of 0.001. The number of vulnerable bits grows in accordance to Moore’s law. As we can observe from the figure, the IBM goal of 114 FIT can be achieved in year 2007 with 100% AVF with 80% of its bits protected; and in 2008 with 10% AVF and no error protection. However, with 10% AVF in 2007, we don’t need to incorporate any error protection at all, to meet the 114 FIT target. Similarly, in 2013, this goal can be met with 80% bits protected against errors, if the AVF is reduced to 10%. This underscores the importance of reducing the AVF, and its impact in the error protection features of a processor.

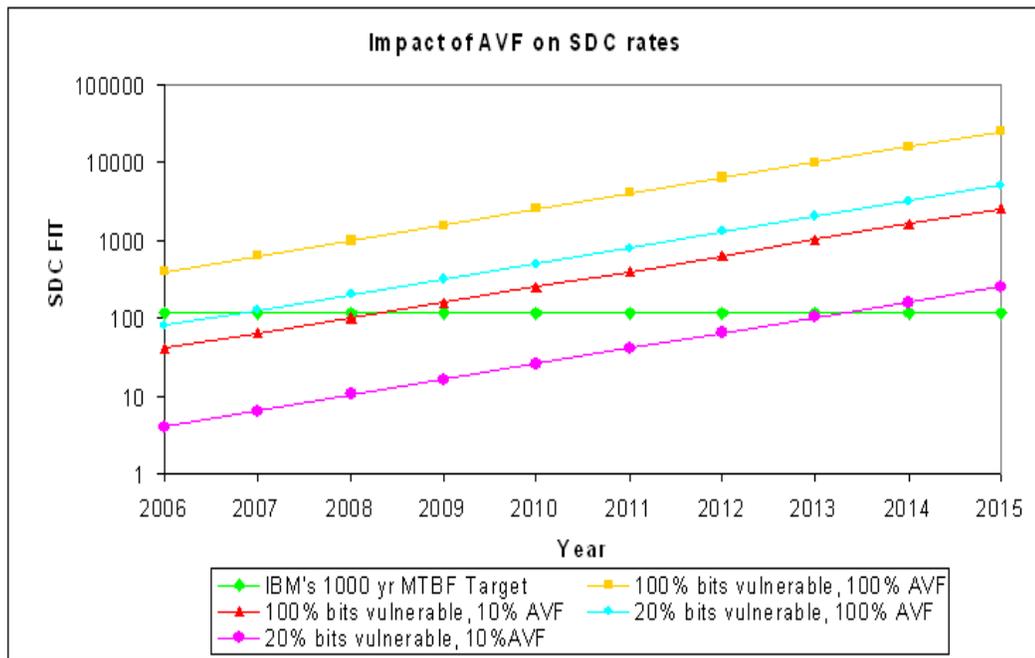


Figure 3.1: Impact of AVF on SDC Rates

For our work, we chose to fully investigate the Instruction Issue Queue because the AVF of this structure was observed to be greater than that of most of the other structures, and hence is one among the highly vulnerable hardware structures on-

chip. An evaluation of the AVF of several microarchitectural structures has been reported previously by the research community [6], [25]. The AVF of the register files was reported as 18.65% by Reis et al [25] and those of the ROB and functional units were estimated as 28% and 9% respectively by Mukherjee et al [6]. In our work, we estimated the AVF for the Instruction Issue Queue as 32.7%, which is significantly higher than the AVFs of the other microarchitectural structures mentioned above.

The next question that needs to be answered is *how* to reduce the AVF of a hardware structure. As discussed in Chapter 1, physical solutions to the problem of reducing soft error rates are hard. Even though special shielding materials can dramatically reduce the impact of alpha particles on semiconductor devices, several feet of concrete is required to shield neutrons, which is not possible for most applications.

To detect or recover from faults, designers typically introduce redundant hardware. Storage structures such as caches, TLBs and memory are protected by error correction codes (ECC) or parity. However, adding such fine-grained protection to all hardware structures can be prohibitively expensive. Other techniques to tackle this problem include duplicating certain hardware structures such as functional units or processor cores, introducing significant penalty in performance, power, die area and design complexity. We focus on alternative techniques to reduce the soft error rates.

We make the key observation that if we can keep valid bits out of the vulnerable structures, a reduction in the soft error rate is guaranteed. In a modern microprocessor, instructions are often fetched and decoded aggressively, and entered into the

IQ, so that they can be issued as early as possible to achieve maximum possible performance benefits. However, instructions are often un-necessarily fetched much earlier than essential, causing valid instructions to reside in the vulnerable IQ for many needless cycles, rather than in a better-protected Instruction Cache, waiting for dependencies to be resolved or to be aborted following a misprediction event.

In this work, we propose front end throttling techniques, to reduce the residency cycles of a given instruction in the Instruction Issue Queue before issue. By this, we reduce the total time any given bit would be vulnerable to neutron or alpha particle strikes, thereby reducing the AVF and hence the soft error rate. However, this would indirectly imply degradation in performance as well, which is undesirable. We investigated a set of previously proposed fetch/decode gating schemes with appropriate adaptations to the reliability perspective.

3.2 Related Work

In this section, we discuss prior research related to our work, and the significance of our research.

Our work is related to two broad fields of research in computer architecture. The first field is related to measuring the AVF of a typical microprocessor. For their studies, prior research groups in the fault-tolerance area of research mainly used two approaches: the Statistical Fault Injection(SFI) model based approach, or the Performance Model based approach. Our work is based on the latter approach.

In the SFI based approach, detailed RTL based processor models are used.

Faults are intentionally created in the processor with special software and/or hardware tools and the operations are monitored. Kim and Somani [21] measured the soft error sensitivity (SES) of the various hardware structures using the example of Sun Microsystem's publicly disclosed picoJava II RTL model through a software simulated fault injection study. Using SFI, Saggese et al [36] experimentally studied the effects of transient faults on two microprocessors: a DLX processor, representative of a microprocessor for an embedded system, and an Alpha processor, representative of a high-performance microprocessor.

The Intel FACT group used a generalized performance model based approach to compute the architectural vulnerability of various microarchitectural structures [6]. In their work, Mukherjee et al proposed a generalized systematic analysis scheme by which, bits in a structure can be classified into those which are required for correct program execution and those which does not matter. The SFI approach is advantageous over the performance based model since an RTL model of a processor will model in detail, every structure that is required to build a processor; while a performance model usually simulates only those components which affect the processor's performance. This limits the application of the performance model based approach. But, the SFI approach would require a significant number of fault injections to achieve statistical significance. Compared to SFI, the latter approach would give faster and more sophisticated analyses, in a single experiment, providing reasonably tight AVF estimates.

In our work, we used the performance model based approach to quantify the architectural vulnerability of the Issue Queue. Inspired by the work of Mukherjee

et al [6], we developed a methodology to measure the AVF, specifically for the IQ. We discuss this in detail in Chapter 4.

The second field to which our work is related is power-aware front-end throttling techniques. Various fetch/decode throttling schemes have been published by different research groups, specifically to reduce the power dissipated by the Issue Queue. Banisadi and Moshovos [12] studied a set of front-end throttling schemes which make use of the instruction flow information, to employ throttling. In [14], Karkhanis et al proposed a just in time instruction delivery scheme, by dynamically limiting the total number of in-flight instructions. They reported a 40% power reduction in the Issue Queue with only 3% degradation in performance.

We apply techniques which were originally proposed to reduce the power consumption of a microprocessor, to reduce the AVF. Our methods were developed based on the schemes proposed earlier by Banisadi and Moshovos [12], Buyuktosunoglu et al [13], Karkhanis et al [14]. We modified these techniques to suit the reliability context inspired from the work of Balasubramonian et al [20] for finding optimal cache sizes. We implemented *adaptive* schemes with varied throttling levels based on the program behavior, instead of a fixed-threshold based throttling. Most of the front-end throttling techniques target at eliminating un-necessary speculative instructions being fetched into the IQ. Since these wrong path instructions will not contribute towards the AVF of a structure, we do not stand to gain by eliminating the wrong-path instructions alone. However, by using appropriate triggers to initiate the throttling, and by using an adaptive method to suit the various phases of within and across applications, we found that we can significantly reduce the aver-

age number of cycles for which each instruction resides in the IQ before its issue. This, in turn, would serve greatly to bring down the AVF.

Redundancy in the form of software or hardware, have been the most widely proposed solution to enhance the fault-tolerance features of microprocessors. Austin proposed the DIVA design [38] where a main out-of-order processor core executed instructions in the normal fashion, and a second simpler one validated the execution. Oh et al [39] proposed a purely software technique called EDDI(Error Detection by Duplicated Instructions) where each instruction is duplicated during compilation, and these checker instructions executed for validation. Oh et al [40] also developed a software control-flow checking scheme called CFCSS(Control Flow Checking by Software Signatures) where each control flow generates a run-time signature which is validated by error-checking code generated by the compiler for every block. Almost all redundancy based techniques report as much as 98% fault coverage, but accompanied by heavy performance loss.

Weaver et al introduced a different approach to improve the soft-error induced unreliability. In their work [23], Weaver et al introduced methods to distinguish *false* detected errors, which do not influence the final system output, from *true* detected errors to improve the DUE AVF of the system. They reported the improvement in the AVF of the re-order buffer(ROB) on detecting and eliminating the false DUE errors. In addition, they proposed *selective instruction squashing* as a method to reduce undetected SDC errors in the ROB of a simple in-order processor. In this method, on a cache miss, the authors propose to flush out the instructions in the ROB which are younger than the load that missed. Because they examined

an in-order machine, this squashing of all instructions after a load miss had a lesser impact on performance, compared to an out-of-order machine. For the above techniques the authors reported an IPC loss of up to 10%, for a soft error improvement of 35% for SDC errors, when instructions were squashed on load misses from L0 cache.

In addition, we draw on prior work in measuring the architectural behavior of processors, for our ACE analysis. Several publications (e.g., [34], [35]) have reported the percentage of dynamically dead instructions, with modern microprocessors. Butts and Sohi [34] reported 3 to 16% of dynamically dead instructions in their evaluation of SPEC2000 benchmarks. We exploit logical masking of operand bits in our ACE analysis. Previously, Scarbrough et al [37] used logical masking to avoid un-necessary fault-detection checks, in their work.

Chapter 4

ENHANCING RELIABILITY

This thesis aims to enhance the dependability of the Issue Queue, using front-end throttling. For this, we first require a method to quantify the soft error vulnerability of the Issue Queue, in order to estimate the improvement achieved by the proposed methods. No ready-to-use tools are available to measure the AVF of a structure. Hence, this chapter first presents the AVF computation methodology that we have developed to accurately quantify the soft error vulnerability of the Issue Queue. Secondly, we explain in detail, the three different throttling schemes explored to bring down the soft error rates.

4.1 AVF Computation

The AVF of a hardware structure is the average AVF of all the bits in that structure, assuming all bits in the structure have the same structure and circuit composition, and hence the same raw FIT rate.

In order to compute the AVF of the structure IQ, we need to first identify the ACE and un-ACE bits in the structure, i.e., figure out which bits affect the final output of the program. For this ACE analysis, we need to clearly define the term *final output*. We define this term in a general sense as: the output of a program is nothing but the values conveyed by the program to I/O. However, in our work,

we do not track values as far as the I/O. But, we track values well past the commit point, to decide whether they will affect the final output or not. We analyze only a uniprocessor system in this work.

4.1.1 ACE Analysis

Given the above definition of a program's output, we need to categorize the ACE and un-ACE bits in the IQ structure. In our work, we strive to unearth as many sources for un-ACE bits as possible. We believe that we have managed to capture all the major contributors of un-ACE bits.

In our experiments, we followed the conservative path. Unless proven otherwise, we assume that every bit in the structure is ACE. We identified the following classes of un-ACE bits in the IQ entries.

Idle Bits

Idle bits are the ones that do not contain any valid information. In the case of an IQ, several of the IQ slots may be empty at any given cycle. For such cases, only the status bits showing whether the slot is empty/full are ACE bits. If these status bits are flipped, the empty instruction entry may get issued and the final output will be affected. All other bits are classified as un-ACE. We always assume the control bits as ACE.

Wrong Path Bits

At any time, there may be several instructions that are executed in a speculative fashion. Branch prediction and speculative loads are examples of this. These instructions reside in the IQ for several cycles, and later a subset of these may get squashed after being discovered to be in the incorrect path. The bits in the IQ corresponding to an incorrectly speculated instruction are un-ACE.

Dynamically Dead Bits

The instructions whose results do not get used are said to be *dynamically dead*. Instructions whose results do not get read by any other instructions at all are termed as First-Level Dynamically Dead (FDD) instructions. The results of Transitively Dynamically Dead (TDD) instructions are read only by FDD instructions or other Transitively Dynamically dead instructions. Instructions can be dynamically dead through registers or memory, depending on whether the result is written to an architectural register or to a memory location. For example, consider an instruction (I1), which writes a value into the register R1. If another instruction (I2) writes into the same register R1, without any intervening reads between I1 and I2, then I1 is and FDD instruction (through register R1). Even if I1 gets a fault, and is executed incorrectly, it does not matter as far as the final outcome is concerned. In a similar fashion, we can define FDD instructions with respect to memory. If I1 writes to a memory location M, and I2 over-writes M, before any other instruction reads M, then I1 is FDD through memory. The only bits that matter in a dynamically dead

instruction would be the destination specifier bits. If the destination gets changed due to a fault, the final output may be affected. In our experiments, we track down both FDD and TDD instructions through registers as well as memory. We assume that the opcode bits and the destination register/memory specifier bits as well as any control bits as ACE bits. The rest of the bits in a dynamically dead instruction are classified as un-ACE.

Unused Bits

There are several instances when bits within valid instruction entries are not used. In an IQ slot occupied by a valid instruction, not all bits are required for architecturally correct execution. For example, if the instruction is a NOP, all bits other than the opcode bits are un-ACE. The operand value bits of an instruction whose operands are not ready will be *unused* until they are ready. We term such bits which do not matter within a valid instruction entry as *unused bits*.

Performance Enhancing Instruction Bits

Modern ISAs include performance enhancing instructions like pre-fetch. A bit flip in non-opcode fields of these instructions, while degrading performance, would not affect the correctness of the program. Hence we consider the non-opcode bits of such instructions as un-ACE.

Masked Bits

Another source of un-ACE bits would be masked bits in operands. Consider the following instruction:

```
ORI $R1, $R2, 0x00FF
```

Here, the lower 8 bits of R2 does not play a part in the result.

Consider another instruction as below:

```
ANDI $R1, $R2, 0x00FF
```

Here, all bits except the lower 8 bits of R2 do not influence the final result. The bits which do not affect the final result are said to be *masked* and hence un-ACE.

In the above analysis, we assume that the user does *not* run the program under a debugger. In such a case, even the intermediate program variables examined by the debugger will count as outputs.

4.1.2 Computing AVF using a Performance Model

The AVF of a single bit storage cell is the fraction of time it holds ACE bits. If a storage cell holds ACE bits for 5 million cycles out of a total of 10 million execution cycles, then its AVF is 50%. We can generalize this notion to the whole structure. The AVF of the full structure is the sum of the contributions from its individual bits. We assume that all the bits in a structure are similar, having the same circuit composition and hence the same raw error rate. The AVF of a hardware structure is given by [6] as:

$$\frac{\text{sum of the residency (in cycles) of all ACE bits in the structure}}{\text{total number of bits in the hardware structure} \times \text{total execution cycles considered}}$$

In order to compute the AVF using the above equation, we need to measure the following parameters: the number of cycles for which each instruction resides in the IQ before its issue, the number of ACE bits in the IQ entry corresponding to each instruction, the number of various classes of un-ACE bits in the IQ entry for each instruction fetched, the total number of bits in the IQ and the total number of execution cycles considered. Using a performance model, we can measure all the above. As an instruction flows through the pipeline, we measure the residence cycles of the instruction at each stage using a performance model.

In order to estimate dynamic deadness, we need information about the future use of the instruction. For this, if the instruction commits, we enter it into an analysis window. We estimated that, a size of 40000 instructions for the analysis window would suffice to capture much of the needed information to estimate deadness. For each architectural register/memory location, we maintain a *usage list* which is a linked list of producers and consumers in the commit order. As soon as an instruction enters the analysis window after commit, we update the producer-consumer list. If a register R has two consecutive producers in its usage list, then the first producer instruction is marked dynamically dead(FDD). Transitive dynamic deadness is estimated similarly from the usage list.

4.2 Reliability-aware Front-end Throttling

In our work, we introduce a set of reliability-aware front-end throttling methods. These methods rely on the well-understood inefficiency of conventional front end instruction delivery designs. Existing designs process instructions at the maximum possible rates to exploit as much parallelism as possible. Additional ILP that may be exposed (if at all) by the aggressive instruction fetch and decode, does not always improve performance. Often times, the processor is stalled waiting for some other instructions to complete. We exploit this inefficiency by slowing down the front end instruction delivery when it is not likely to affect performance significantly. Instructions are fetched *just-in-time* to exploit the parallelism.

The idea of front-end throttling is not new. However, traditionally, these techniques are employed largely for on-chip power-consumption reduction. But, as one can clearly derive, throttling of the front-end of the pipeline would reduce the number of cycles for which each instruction resides in the IQ before issue, which in turn serves to directly reduce the *vulnerability* of the structure.

As we know, different applications behave differently, and even a single application has distinct phases differing in behavior, throughout its execution. Because of this, we cannot expect the same level of throttling to work for all applications, or even for multiple phases of a single application. Hence, we implement *adaptive* front end throttling schemes in which the throttling level is changed at fixed intervals, depending upon the behavior of the application. In each of the methods implemented, we choose a suitable *trigger* based on the value of which the decision to throttle is

made. We maintain a dynamic *threshold* for this trigger value, which specifies the maximum value that the *trigger* can take at any point of time. At any cycle, if the *trigger* surpasses its *threshold* value, we throttle the front end. We dynamically tune for the least value of this *threshold* such that the performance degradation does not exceed 2%. To perform this dynamic tuning, we use an algorithm derived from the one proposed by Balasubramonian et al [20] for finding optimal cache sizes. The entire program execution is divided into *intervals* (100K cycles in our simulations). At the end of each *interval*, a hardware counter is examined which gives the IPC experienced by the application in the past interval. Based on this information, the *threshold* value is updated. The decision to throttle the front-end is done on a cycle-by-cycle basis.

The selection mechanism for the *threshold* is as follows. For the very first interval, we let the application run with the *threshold* set to the maximum possible value, i.e., without any throttling employed. We record the IPC during this interval as our *base IPC*. For the next interval, the *threshold* is set to the minimum possible value, i.e., with maximum throttling employed. In subsequent intervals, *threshold* is incremented by a fixed quantity called a *step* until the performance for an interval falls within 2% of the base IPC. We call this as a *tuning period*. After this tuning period, the threshold is kept at this *optimum* value. The base IPC is refreshed once in every *window* of 20 intervals. This is done by resetting the threshold back to the maximum value once in each window of 20 intervals and repeating the process mentioned above. Typically a phase change occurs only once in several 100's of intervals or more. During any cycle, if the *trigger* value exceeds the set *threshold*,

we disable the fetch and decode stages for that cycle.

We chose an interval size of 100K cycles so as to react quickly to changes without letting the selection mechanism pose a high cycle overhead. We chose a moderate window size of 20 intervals to ensure that any phase changes are captured quickly enough. Since, a phase change can occur very early in a window, if the window-size is set too high, the degradation in IPC becomes correspondingly higher. With a window size of 20 intervals, we spend 5% of the total execution cycles (1 out of every 20 intervals) in capturing the program behavior and the remaining 95% in optimizing the AVF.

We simulated the following three schemes of front end throttling, as described below, to reduce the AVF of the IQ.

- (1) Occupancy based Throttling
- (2) Flow based Throttling
- (3) Rate based Throttling

4.2.1 Occupancy based Throttling

Intuitively, the most direct metric to use as a trigger to employ throttling would be the occupancy of the Issue Queue itself, since we are aiming at reducing the IQ utilization. Hence, in our first scheme, the front end of the pipeline is throttled based on the number of instructions residing in the IQ, during each cycle. Hence, the occupancy of the issue queue is the *trigger*, and the maximum IQ occupancy possible (*MAX_Occupancy*) as our *threshold*. Initially, *MAX_Occupancy* is set to the orig-

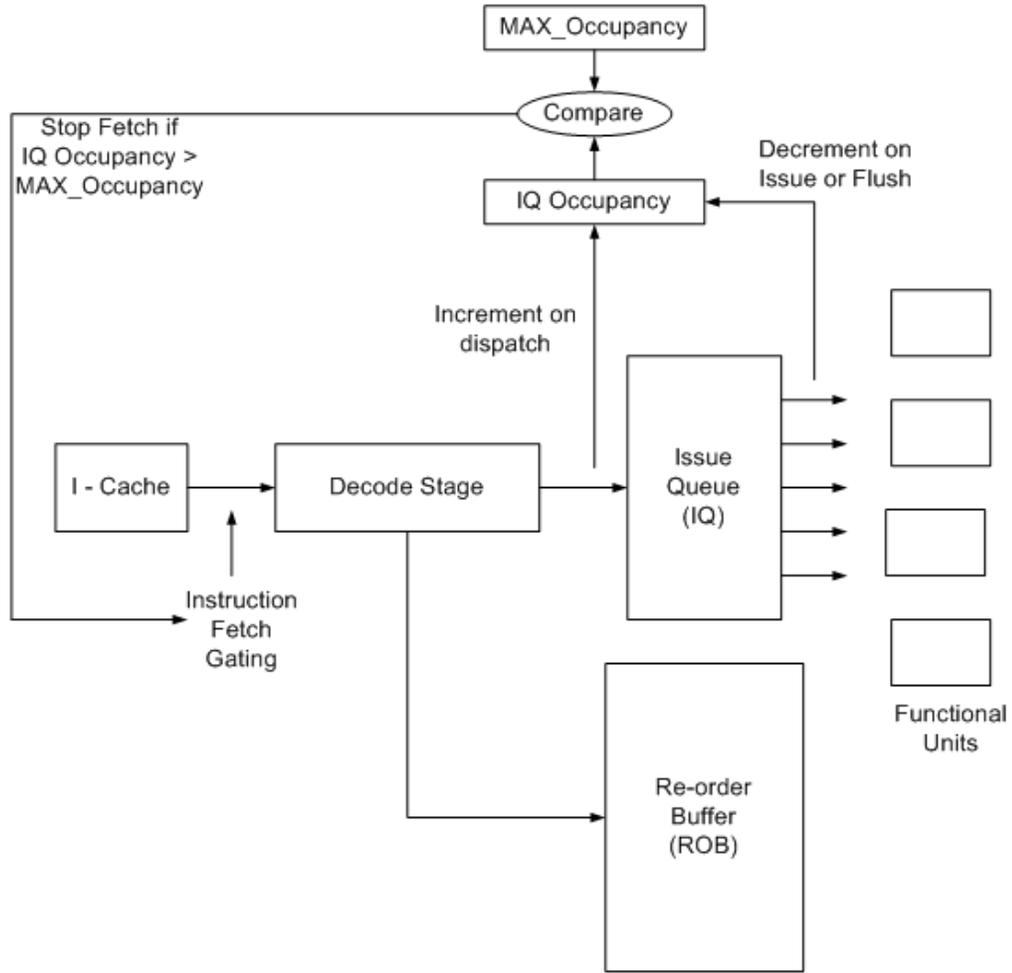


Figure 4.1: Pipeline with Control Logic for Occupancy Based Throttling

inal IQ Size, which is the maximum value possible for the IQ occupancy (32 entries in our case). As elaborated earlier, the *base IPC* is recorded after the first interval. In the following interval, we set the minimum possible *MAX_Occupancy*, which we set as 10% of the original IQ Size. In subsequent intervals, *MAX_Occupancy* is increased by *steps* of 10% of the IQ size until the performance of a particular interval comes within 2% of the *base IPC*. *MAX_Occupancy* is then held at this *optimum* value until a *window* of 20 intervals expires. Once the *window* expires, we resume

the dynamic tuning, and repeat the process. The decision to throttle the front-end is done on a cycle-by-cycle basis. If, in a particular cycle, the IQ occupancy exceeds the *MAX_Occupancy* threshold value, fetch and decode stages are disabled in that cycle.

4.2.2 Flow based Throttling

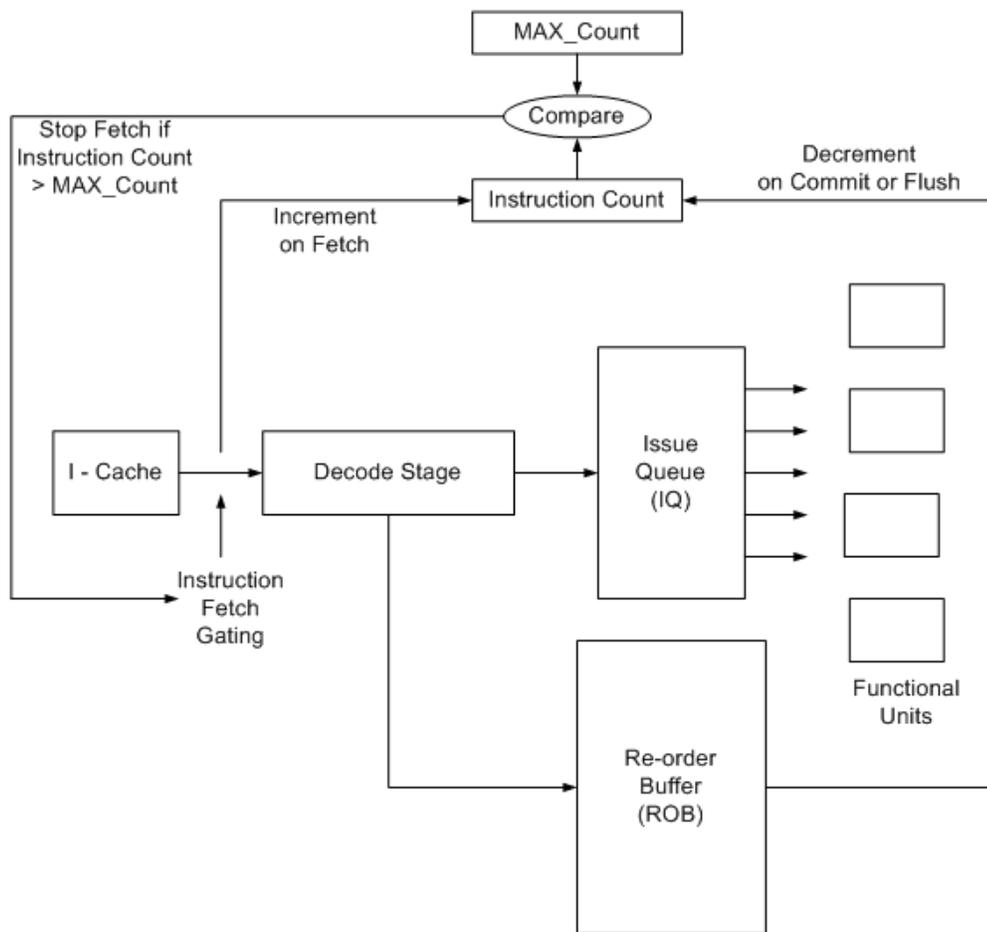


Figure 4.2: Pipeline with Control Logic for Flow Based Throttling

With this technique, the *trigger* employed is the total number of *in-flight instructions*, during each cycle. An instruction is said to be *in-flight* after it is

decoded and before it is committed. This method is based on the observation that, increasing the number of active instructions in the pipeline beyond a limit, does not offer any significant performance benefits. Similar to the previously discussed method, the *threshold* value is the maximum number of in-flight instructions possible (*MAX_Count*). The *MAX_Count* is initially set to the maximum possible value (256 in our case). The IPC is recorded after one 100K cycle interval as the *base IPC*. For the next interval, *MAX_Count* is set to its minimum possible value, which is 16 instructions. In subsequent intervals, *MAX_Count* is increased by *steps* of 16 instructions each until the performance for an interval falls within 2% of the *base IPC*. *MAX_Count* is held at this *optimum* value until a *window* of 20 intervals expire. Once the *window* expires, we reset the *MAX_Count* to its maximum value of 256 and resume the dynamic tuning. The decision to throttle is done on a cycle-by-cycle basis. If the total number of in-flight instructions in a particular cycle exceeds the threshold value of *MAX_Count*, the front-end is throttled for that cycle.

4.2.3 Rate based Throttling

We observe from wide-issue superscalar simulations that in program phases where instruction level parallelism (ILP) is limited, there is often a significant mismatch between the front-end decode rate and the back-end retirement rate, leading to an un-necessarily high utilization of the issue queue. In the rate based throttling method, we estimate that sufficient amount of parallelism exists when the number of

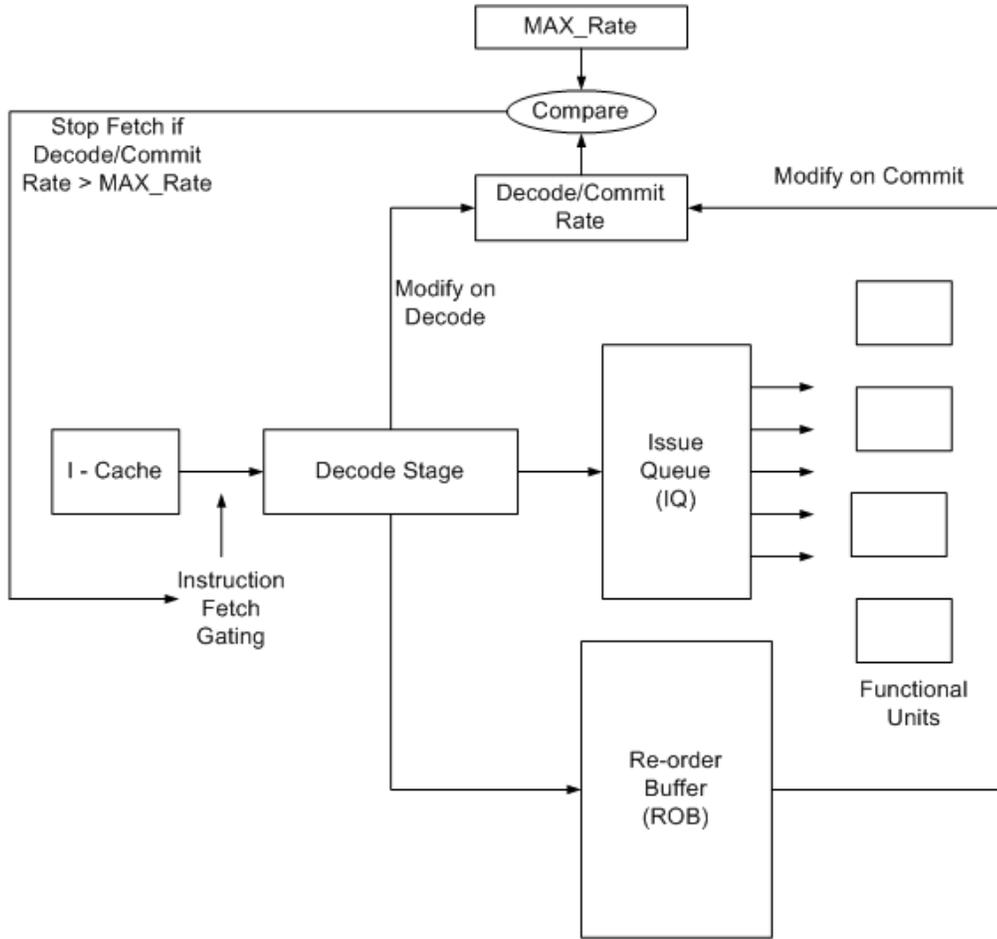


Figure 4.3: Pipeline with Control Logic for Rate Based Throttling

instructions that are decoded exceeds significantly the number of instructions that are committed. We slow down the front end based on the Decode-rate to Commit-rate ratio mismatch, on a cycle by cycle basis. In this method, our *threshold* value is the maximum ratio of decode to commit rate (MAX_Rate). Initially, the MAX_Rate is set to the maximum possible value (4 in our simulations). The IPC is recorded after one 100K cycle interval as the *base IPC*. In the following interval, the MAX_Rate is set to minimum, which is 1, since a decode-to-commit ratio less than 1 is sure to stall the execution. In subsequent intervals, MAX_Rate is

increased by *steps* of 0.5 until the performance for an interval falls within 2% of the *base IPC*. *MAX_Rate* is set at this *optimum* value until a *window* of 20 intervals expire. Once the *window* expires, we reset the *D-C Rate* to the maximum value and perform the dynamic tuning again.

4.3 Overhead Incurred

We propose that the selection mechanism be implemented in software. After every *interval* of 100K cycles, a low-overhead software handler will be invoked that examines the necessary hardware counter holding the IPC of the previous interval and updates the *threshold* value as dictated by the specific algorithm. For this, the hardware overhead imposed is negligible. At the same time, the software implementation allows flexibility in terms of modifying the selection mechanism. We estimate a code size of only a few tens of static assembly instructions for the software handler. This amounts to only a fraction of the instructions executed during each interval between successive invocations of the handler, resulting in a net overhead of around 0.1% or less. In terms of hardware overhead, we require roughly 4 20-bit counters to keep track of the number of instructions committed, the number of cycles elapsed, the number of intervals elapsed and the trigger value. Additional registers are required to store intermediate values such as the threshold and the base IPC. The net overhead amounts to less than 5000 transistors.

Chapter 5

EXPERIMENTAL RESULTS

5.1 Simulation Model

For our microarchitectural simulations, we used a modified version of the SimpleScalar 3.0 simulator [10], [11]. We modified the simulator to model the front end instruction delivery in more detail, and with better accuracy than in the baseline simulator. The Instruction Issue Queue (IQ) was modeled as a separate structure from the re-order buffer (ROB). We use fetch, decode and issue widths of 6, 6 and 4, respectively. Though these lower numbers for the machine widths served to diminish the relative improvements in soft error rates to some extent, we believe they present more realistic machine widths compared to wider machines assumed for related work [29]. Table 5.1 summarizes the processor configuration used in all simulations. We use a number of integer and floating point benchmarks from the SPEC2000 suite.

We implemented the IQ as a circular FIFO queue without any collapsing. That is, if an instruction in the middle of the IQ is issued, the corresponding entry becomes empty. Even though collapsing makes more effective use of the instruction queue, it is much more energy-consuming since collapsing implies a shift of all the entries between the empty entry and end of the IQ, and hence not preferred.

For our simulations, we used a subset of the SPEC2000 integer and floating point benchmarks (gcc, gzip, vpr, mcf, bzip2, art, earthquake, mesa and swim). Table 5.2

Table 5.1: Processor Configuration

PARAMETER	VALUE
ROB Size	256 entries
IQ Size	32 entries
LSQ Size	128 entries
Fetch Width	6 instructions/cycle
Decode Width	6 instructions/cycle
Issue Width	4 instructions/cycle
Commit Width	4 instructions/cycle
Branch Predictor	Combined predictor : bimodal (8K entry) + 2-level adaptive (8K entry), 8K meta predictor
Functional Units	4 Integer ALUs 1 Integer Mult/Div unit 4 FP ALUs 1 FP Mult/Div unit
L1 cache : Split I and D caches	64KB, 2 way associative (each)
L2 cache : Unified	1MB 4 way associative

lists the skip interval and input set selected for each of the benchmarks chosen. We obtained the skip interval using the SimPoint Analysis [26], [27]. For each benchmarks, we obtained a number of simpoints. The numbers presented are for the first simpoint of each benchmark. For each simpoint, we executed 100million instructions (excluding NOPs).

Our calculations assume each entry of the instruction queue to be approximately 160 bits. The number of bits required per IQ entry is always much higher than the number of bits in the instruction itself. This is because a large number of

Table 5.2: SPEC2000 Benchmarks used and Instructions skipped.

Benchmark	Instructions skipped (in millions)
gcc-166	400
gzip-graphic	39000
vpr	57000
mcf	28300
bzip2-graphic	49400
art	39000
equake	16800
mesa	76000
swim	96600

additional bits are required to capture the state of the in-flight instruction.

In our bit-level ACE analysis, we assume all control bits in the IQ entries to be always ACE bits. Also, for all valid IQ entries, the eight opcode bits are always assumed ACE. This is because any flip in the opcode bits will cause the instruction to be interpreted wrongly and might affect the output. For the dynamically dead instructions, the five destination register specifier bits are also considered ACE, apart from the opcode specifier bits and the control bits.

5.2 AVF of the Instruction Queue

This section presents the results of the first phase of our work - the quantification of the AVF of the Instruction Issue Queue (IQ). According to the methodology

mentioned in Chapter 4, we classified the bits that occupy the IQ into several categories. We find that the AVF of the IQ ranges between 18% and 49%. We observe that Floating Point programs, in general, exhibit a higher AVF (35%) in comparison with Integer Programs (30%). In general, floating point programs have a large number of long-latency instructions and fewer branch mispredictions. This leads to a greater instruction queue utilization for floating point programs and hence a higher AVF.

Figure 5.1 shows the split-up of the total bits in the IQ. We observed 32.7% of ACE bits, 33.8% idle bits, 4.9% mis-speculated instruction bits, 18.9% of un-used bits in the IQ slots which does not matter, 0.8% of masked operand bits and 8.9% of dynamically dead instruction bits.

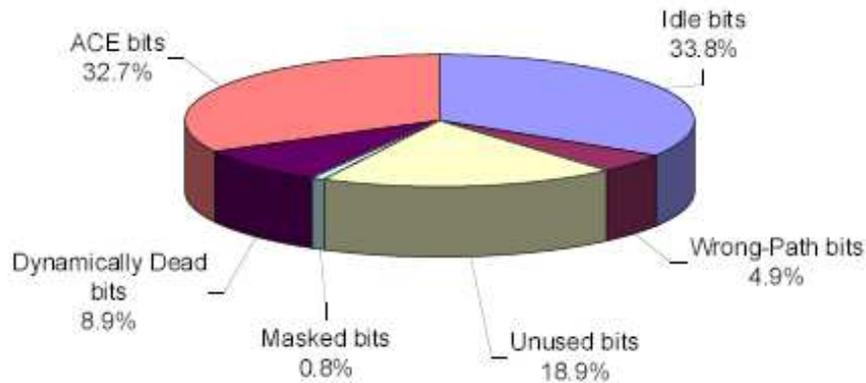


Figure 5.1: Bit-level Classification of the IQ Contents

5.3 Effects of Front-End Throttling

The effects of the three schemes of front-end throttling on (i) the average number of cycles that an instruction resides in the IQ before it is issued, (ii) performance, (iii) AVF and (iv) MITF are discussed in this section.

5.3.1 Reduction in the Residency Cycles in IQ before Issue

Figure 5.2 shows the variation in the average number of cycles that an instruction resides in the Instruction Queue before issue. Since the intuition behind our work was to deliver the instructions into the IQ just in time for issue, reduction in the residency cycles in the IQ before issue is a direct pointer to the validity of our reasoning.

On average, an instruction resided in the IQ for 8.52 cycles before issue without any throttling employed. On employing the Instruction Decode-Retirement Rate based throttling scheme, there was a 13.4% reduction in this value. The residency cycles in the IQ reduced to 7.40 cycles. The In-flight instruction Flow based throttling and IQ Occupancy based throttling schemes yielded 18% and 24.3% improvements in the same, reducing the average number of residency cycles to 6.99 and 6.44 respectively.

5.3.2 Improvement in AVF

Figure 5.3 depicts the relative improvements in the architectural vulnerability factor due to the different throttling methods explored. Without any throttling,

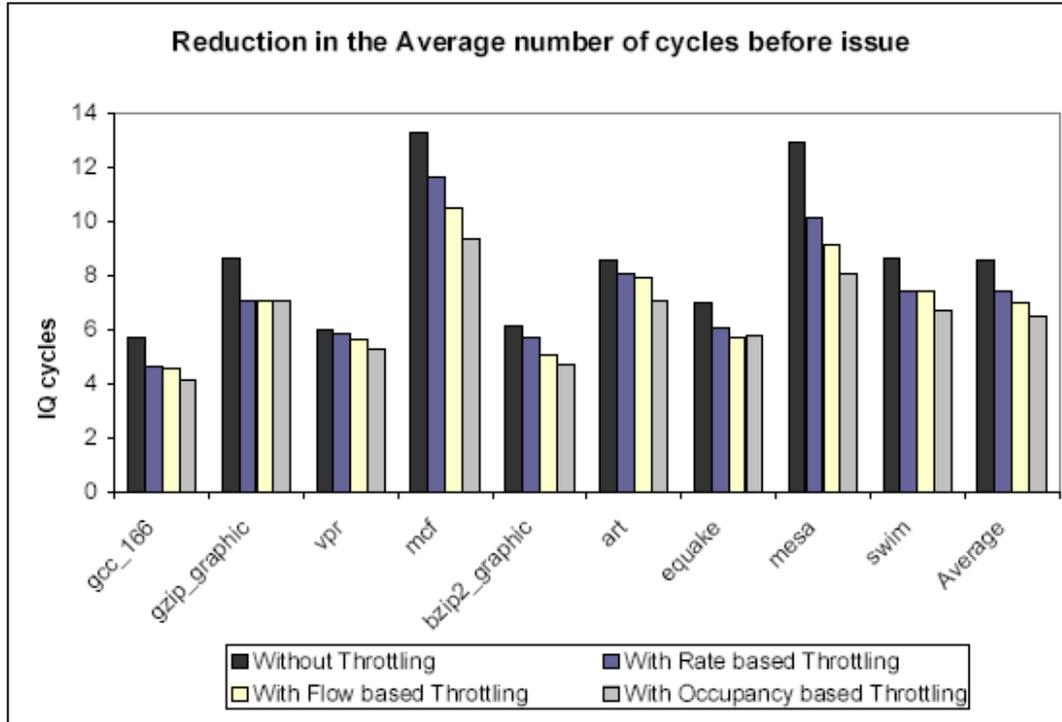


Figure 5.2: Effect on the Average Number of Residence Cycles in IQ before Issue

we measured the original AVF of the instruction queue as 32.7%. By employing the rate-based, flow-based and occupancy-based throttling schemes, we were able to reduce the AVF figure to 27.4%, 25.8% and 24.1% respectively. Thus we obtained an improvement in the AVF by 16.5%, 21.1% and 26.3% for the Decode-Commit Rate based throttling, In-flight instruction flow based and IQ occupancy based throttling respectively.

5.3.3 Degradation in Performance

Figure 5.4 depicts the degradation in performance as a result of the throttling schemes employed. The maximum degradation in performance was observed with

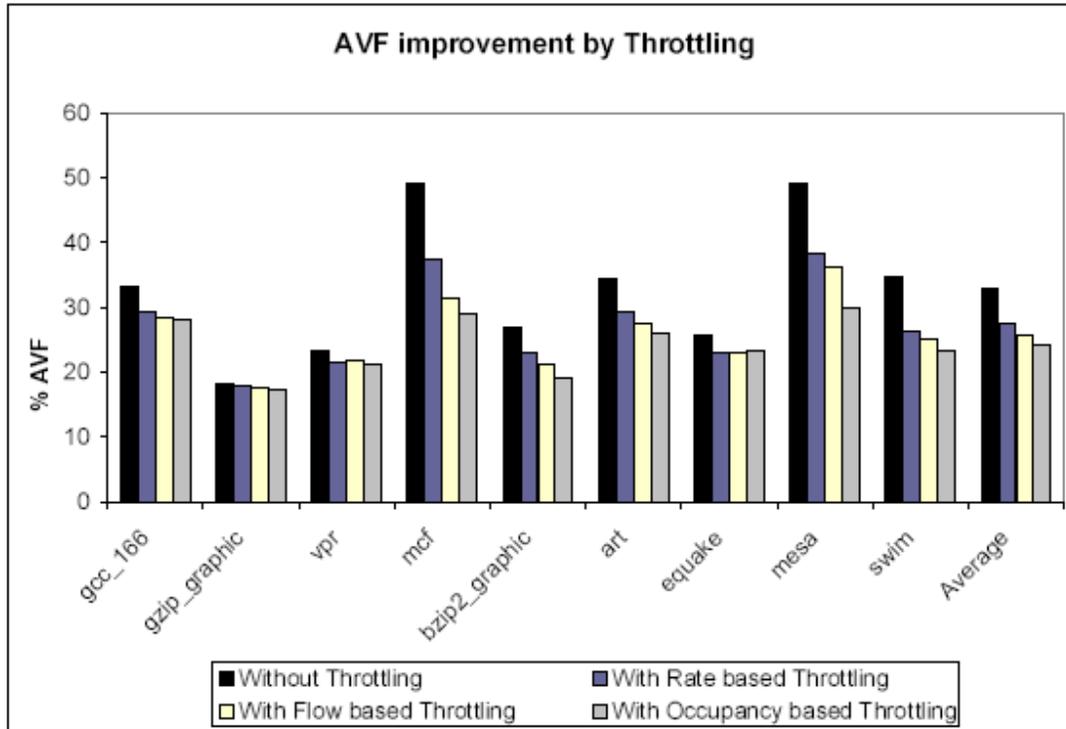


Figure 5.3: Improvement in AVF

the IQ Occupancy based throttling scheme. With this method, the average IPC degradation of 2.6% was observed. The maximum degradation was recorded for the benchmark *gcc* as 3.6%. For the In-flight Instruction flow based throttling, an average IPC reduction of 2.1% was recorded. The maximum degradation occurred was 2.6%. For the Decode-Retirement Rate based throttling, we observed an average performance degradation of 1.6%. A maximum degradation of 2.2% was recorded for this scheme.

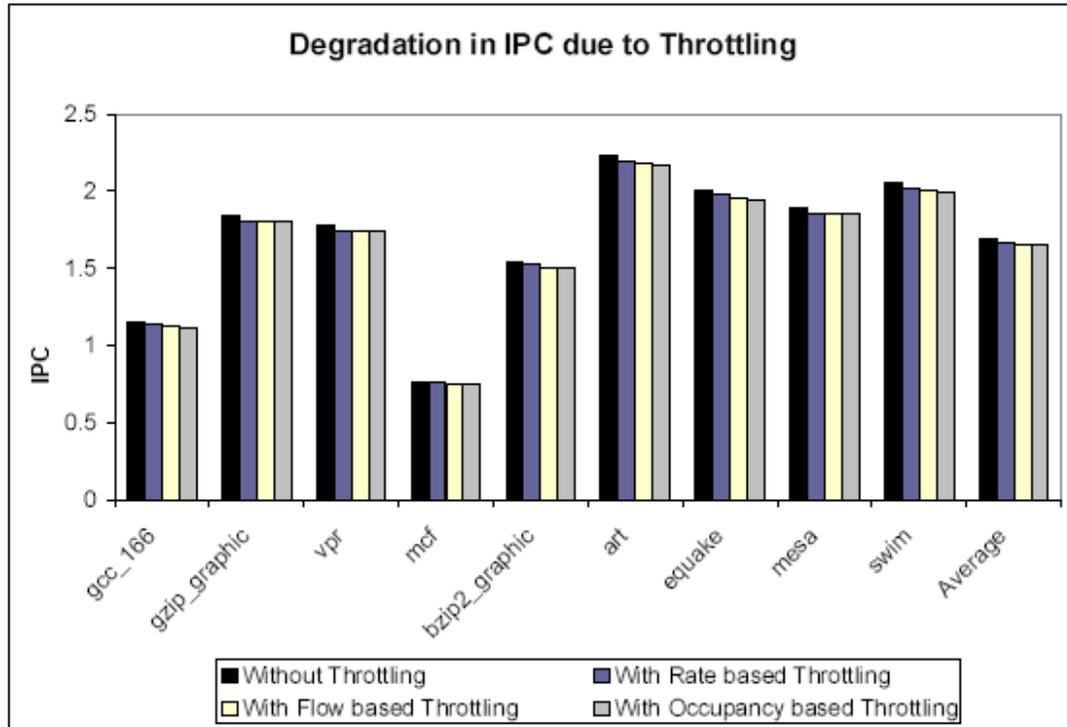


Figure 5.4: Degradation in Performance

5.3.4 Reliability-Performance Trade-off

The Mean Instructions to Failure (MITF) figures gives us the reliability-performance trade-off. Assuming a constant value for the frequency of operation and the raw error rate, we figure out the improvement in MITF as the improvement in the ratio $\frac{IPC}{AVF}$. Figure 5.5 shows the relative improvements in MITF due to the three schemes of front-end throttling. We observed 13%, 16.3% and 22.7% increase in the $\frac{IPC}{AVF}$ and hence the MITF due to Decode-Retirement based throttling, In-flight instruction limit throttling, and IQ occupancy based throttling respectively.

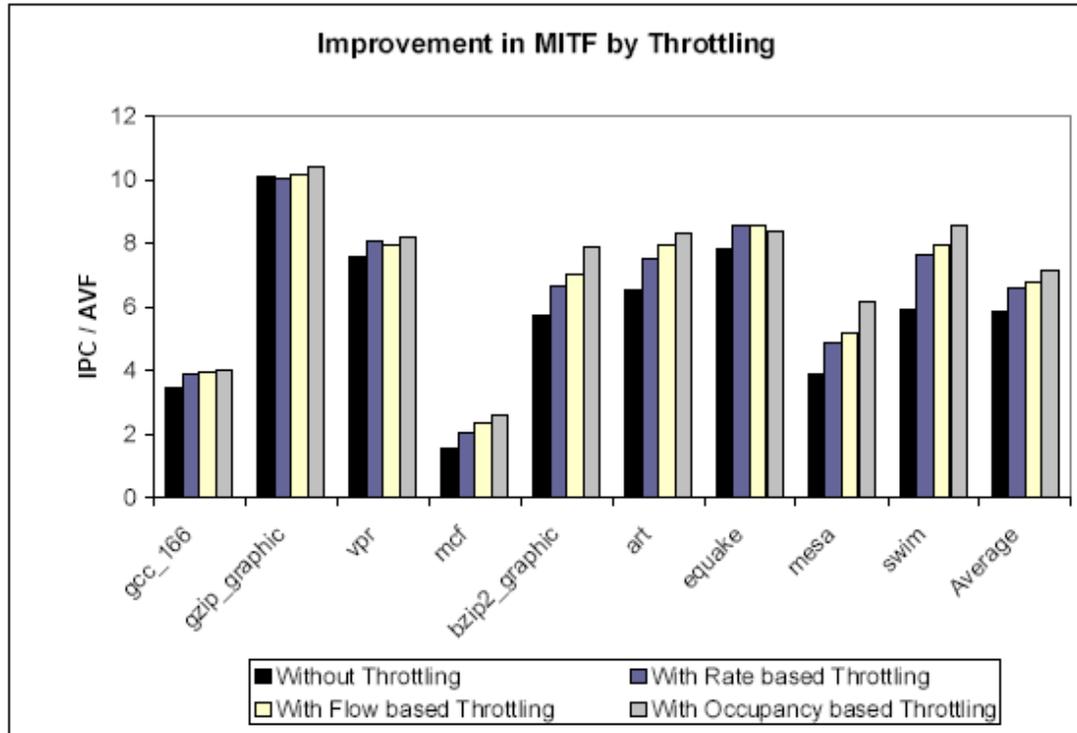


Figure 5.5: Improvement in MITF

5.4 Observations

The front-end throttling methods were observed to result in an increase in the number of idle bits in the IQ, and to a lesser extent, reduction in the wrong-path instruction bits. The increase in the idle bits aids the reduction of the AVF, whereas the reduction in the wrong-path bits serves to decrease the total number of un-ACE bits and hence increase the AVF. The throttling methods in the power-reduction research area, aims mainly to reduce the speculative instructions being fetched. However, in our case, the reduction in the wrong path bits serves to increase the AVF, since these bits are already categorized as un-ACE. The reduction in the wrong path bits was highest for the Decode-Commit rate based throttling technique.

For this method, we observed the number of wrong path bits to be 2.9% of the total bits, where-as for the flow based and occupancy based throttling, the figures were 3.6% and 3.9% respectively. The contribution of idle bits increased from 33.8% to greater than 38% for all three cases. This substantiates the achievement of our goal of reducing the IQ utilization, with minimal performance impact.

From the results of our experiments, we observe that the IQ Occupancy based throttling gives the best improvement in AVF, without significant degradation in the performance, as is made clear by the MITF figures. We reason that this is due to the finer-grained throttling employed with this method. Since the maximum IQ occupancy possible is 32 in our experiments, we get to establish the throttling threshold very near to the ideal value for each phase in different applications. Whereas, in the case of in-flight instruction flow based throttling, we can have a maximum of 256 in-flight instructions. Due to this, we had to opt for a coarse-grained step of 16 instructions, in our algorithm. A finer-grained control in this method increases the tuning period beyond our chosen instruction *window* length, and this was observed to negatively affect the results. The case is similar with the Decode-Retirement rate based throttling. In this method, we observe the highest reduction in incorrectly speculated instructions. While this serves greatly for other purposes like power reduction, this does not help our cause of reducing the AVF of the IQ. Since we already classify any wrong path instructions as un-ACE already, the reduction in wrong path instructions serves to increase the AVF.

Chapter 6

SUMMARY AND CONCLUSION

6.1 Summary

Transient faults due to energetic particle strikes have become a key challenge in modern microprocessor design. Existing techniques to counter these faults come at the cost of significant penalties in power, performance, die area and design time. Even though the soft error rates of individual transistors are projected to remain as roughly the same for the next several technology generations, the overall per-chip fault rates will continue to increase exponentially in accordance to Moore's law. As a result, even logic elements, which were not a great concern in the reliability perspective earlier, have become a major source of concern. In this thesis, we address the question of bringing down the soft error rates in a cost-effective fashion.

Soft errors in microprocessors can be categorized as Silent Data Corruption (SDC) and Detected Recoverable Errors (DUE). SDC errors occur when a structure does not have any form of error detection or correction features. In such a case, a soft fault would go undetected, and can potentially cause errors in the final output. On the other hand, DUE errors happen, when the structure has only error detection features. Error detection features do not serve to decrease the soft error rates, but can ensure a clean output by providing a fail-stop behavior. Finally, if the structure has error correction features, any fault that might happen will get corrected in

time, preventing the manifestation of errors. The overall error rate of a processor is given by the sum of its SDC and DUE error rates. The SDC and DUE error rates of the processor are in turn, the sum of the contributions by its individual components. In our work, we focused on improving the SDC error rate of a processor by targeting logic structures, since most of today’s processors have their memory structures well protected. We concentrated on the structure with the highest SDC rate - the Instruction Issue Queue (IQ).

In this paper, we introduced a simple approach to reduce the SDC rate of the IQ, and hence the overall soft error rates of a microprocessor. We reduced the amount of time for which a valid instruction sits in the IQ before it issue, by disabling the fetch and decode stages whenever we determine that fetching and dispatching more instructions into the IQ will not hold any significant performance advantage. We reason that a fault is less likely to occur in a structure, if it holds fewer valid bits. We implemented our idea of front-end throttling with three different parameters as the basis of the throttling decision: the occupancy of the IQ, the total number of in-flight instructions and the decode-to-commit rate mismatch.

6.2 Conclusion

We proposed a novel cost-effective approach to reduce the SDC AVF and hence the soft error rates of an Instruction Issue Queue. We exploit the design inefficiency of the front end of the pipeline, by stalling the fetch and decode stages, whenever the IPC will not be affected significantly. This method, traditionally used as

a power-reduction technique among the computer architecture research community, was observed to reduce the average time that an instruction sits in the vulnerable instruction issue queue before its issue by up to 24.3%. Since this caused performance degradation to a certain extent, we estimated the trade-off between performance and reliability. The improvement in the performance to reliability (IPC/AVF) and hence the Mean Instructions to Failure (MITF) was upto 22.7%. We observed a reduction in the SDC AVF, and hence the SDC error rates of the IQ by upto 26.3% using our schemes. The proposed method is cost-efficient as it does not incur significant hardware or software overheads. The average performance penalty recorded was only at most 2.6%.

BIBLIOGRAPHY

- [1] T. Karnik, B. Bloechel, K. Soumyanath, V. De, and S. Borkar, *Scaling trends of Cosmic Rays induced Soft Errors in static latches beyond 0.18*, Symposium on VLSI Circuits Digest of Technical Papers, 2001.
- [2] Mukherjee, S.S. Emer, J. Reinhardt, S.K., FACT Group, Intel Corp., Hudson, MA, USA; *The soft error problem: an architectural perspective*, High-Performance Computer Architecture (HPCA-11), 2005.
- [3] Seifert, N.; Xiaowei Zhu; Massengill, L.W.; *Impact of scaling on soft-error rates in commercial microprocessors*, Nuclear Science, IEEE Transactions on, Volume 49, Issue 6, Part 1, Dec. 2002 Page(s):3100 - 3106
- [4] Hazucha, P.; Karnik, T.; Maiz, J.; Walstra, S.; Bloechel, B.; Tschanz, J.; Dermer, G.; Hareland, S.; Armstrong, P.; Borkar, S.; *Neutron soft error rate measurements in a 90-nm CMOS process and scaling trends in SRAM from 0.25- μm to 90-nm generation*, Electron Devices Meeting, 2003.
- [5] Normand, E.; *Single Event Upset at Ground Level*, Nuclear Science, IEEE Transactions on, Volume 43, Issue 6, Part 1, Dec. 1996 Page(s):2742 - 2750
- [6] S. S. Mukherjee, C. T. Weaver, J. Emer, S. K. Reinhardt, and T. Austin, *A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor*, 36th Annual International Symposium on Microarchitecture (MICRO), December 2003.
- [7] J.F. Ziegler, et al., *IBM experiments in soft fails in computer electronics (1978 - 1994)*, *IBM Journal of Research and Development*, pp. 3 - 18, Volume 40, Number 1, January 1996.
- [8] Ando, H.; Yoshida, Y.; Inoue, A.; Sugiyama, I.; Asakawa, T.; Morita, K.; Muta, T.; Motokurumada, T.; Okada, S.; Yamashita, H.; Satsukawa, Y.; Konmoto, A.; Yamashita, R.; Sugiyama, H.; *A 1.3 GHz fifth generation SPARC64 microprocessor*, Solid-State Circuits Conference, 2003.
- [9] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, third edition, 2003.
- [10] www.simplescalar.com
- [11] D. Burger and T. M. Austin, *The SimpleScalar Tool Set, Version 2.0*, Computer Architecture News, 25 (3), pp. 13-25, June, 1997.

- [12] A. Baniasadi and A. Moshovos, *Instruction Flow-Based Frontend Throttling for Power-Aware High-Performance Processors*, International Symposium on Low Power Electronics and Design, 2001.
- [13] Buyuktosunoglu, A.; Karkhanis, T.; Albonesi, D.H.; Pradip Bose; *Energy efficient co-adaptive instruction fetch and issue*, Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on 9-11 June 2003 Page(s):147 - 156
- [14] Karkhanis, T.; Smith, J.E.; Bose, P.; *Saving energy with just in time instruction delivery*, Low Power Electronics and Design, 2002. ISLPED '02. Proceedings of the 2002 International Symposium on 2002 Page(s):178 - 183
- [15] T. Calin, et al., *Topology-Related Upset Mechanisms in Design Hardened Storage Cells*, Radiation and Its Effects on Components and Systems, 1997. RADECS 97. Fourth European Conference, 15-19 Sept. 1997.
- [16] R. C. Baumann. *Soft errors in commercial semiconductor technology: Overview and scaling trends*, In IEEE 2002 Reliability Physics Tutorial Notes, Reliability Fundamentals, April 2002.
- [17] M. Franklin, *A study of time redundant fault tolerance techniques for superscalar processors*, dft, p. 207, The IEEE International Workshop on Defect and Fault Tolerance in VLSI Systems, 1995.
- [18] Mohamed A. Gomaa, Chad Scarbrough, T. N. Vijaykumar, Irith Pomeranz, *Transient-Fault Recovery for Chip Multiprocessors*, IEEE Micro, vol. 23, no. 6, pp. 76-83, Nov/Dec, 2003.
- [19] Christopher Weaver, Joel Emer, Shubhendu S. Mukherjee, Steven K. Reinhardt, *Techniques to Reduce the Soft Error Rate of a High-Performance Microprocessor*, isca, p. 264, 31st Annual International Symposium on Computer Architecture (ISCA'04), 2004.
- [20] Rajeev Balasubramonian, David Albonesi, Alper Buyuktosunoglu, Sandhya Dwarkadas, *Memory Hierarchy Reconfiguration for Energy and Performance in General-Purpose Processor Architectures*, micro, p. 245, 33th Annual International Symposium on Microarchitecture (MICRO'00), 2000.
- [21] S. Kim and A. K. Somani, *Soft Error Sensitivity Characterization for Microprocessor Dependability Enhancement Strategy*, Proceedings of the International Conference on Dependable Systems and Networks (DSN), 2002.

- [22] Wang, N.J.; Quek, J.; Rafacz, T.M.; Patel, S.J.; *Characterizing the effects of transient faults on a high-performance processor pipeline*, Dependable Systems and Networks, 2004 International Conference on 28 June-1 July 2004 Page(s):61 - 70
- [23] George A. Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, David I. August, Shubhendu S. Mukherjee, *Software-Controlled Fault Tolerance*, ACM Transactions on Architecture and Code Optimization (TACO) Volume 2 , Issue 4 (December 2005) Pages: 366 - 396.
- [24] William Bryg, Jerome Alabado, *The UltraSPARC T1 Processor - Reliability, Availability, and Serviceability*, Sun Microsystems Inc., UltraSPARC Processors Documentation, Whitepapers, December 2005
- [25] Reis, G.A.; Chang, J.; Vachharajani, N.; Mukherjee, S.S.; Rangan, R.; August, D.I.; *Design and evaluation of hybrid fault-detection systems*, 32nd International Symposium on Computer Architecture (ISCA), June 2005 Page(s):148 - 159
- [26] Sherwood, T.; Perelman, E.; Hamerly, G.; Sair, S.; Calder, B.; *Discovering and exploiting program phases*, Micro, IEEE, Volume 23, Issue 6, Nov.-Dec. 2003 Page(s):84 - 93
- [27] Greg Hamerly, Erez Perelman, Jeremy Lau, and Brad Calder, *SimPoint 3.0: Faster and More Flexible Program Analysis* , Workshop on Modeling, Benchmarking and Simulation, June 2005.
- [28] Folegnani, D.; Gonzalez, A.; *Energy-effective issue logic*, Computer Architecture, 2001. Proceedings. 28th Annual International Symposium on, 30 June - 4 July 2001 Page(s):230 - 239
- [29] Unsal, O.S.; Koren, I.; Khrishna, C.M.; Moritz, C.A.; *Cool-Fetch: a compiler-enabled IPC estimation based framework for energy reduction*, Interaction between Compilers and Computer Architectures, 2004. INTERACT-8 2004. Eighth Workshop on 15 Feb. 2004 Page(s):43 - 52
- [30] Fred A. Bower, Derek Hower, Mahmut Yilmaz, Daniel J. Sorin, Sule Ozev; *Applying Architectural Vulnerability Analysis to Hard Faults in the Microprocessor*, In Proceedings of ACM SIGMETRICS/Performance 2006, June 26-30, 2006.
- [31] Surendra, G.; Subhasis Banerjee; Nandy, S.K.; *Power-Performance Trade-off using Pipeline Delays*, Design Automation Conference, 2004. Proceedings of the ASP-DAC 2004. Asia and South Pacific 27-30 Jan. 2004 Page(s):384 - 386

- [32] Ponomarev, D.V.; Kucuk, G.; Ergin, O.; Ghose, K.; Kogge, P.M.; *Energy-efficient issue queue design*, Very Large Scale Integration (VLSI) Systems, IEEE Transactions on, Volume 11, Issue 5, Oct. 2003 Page(s):789 - 800
- [33] Smolens, J.C.; Jangwoo Kim; Hoe, J.C.; Falsafi, B.; *Understanding the performance of concurrent error detecting superscalar microarchitectures*, Signal Processing and Information Technology, 2005. Proceedings of the Fifth IEEE International Symposium on Dec. 18 - 21, 2005 Page(s):13 - 18
- [34] J. Adam Butts and G.S. Sohi; *Dynamic Dead-Instruction Detection and Elimination*, 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X), Oct 2002.
- [35] Eric Rotenberg; *Exploiting large ineffectual instruction sequences*, Technical Report, NC State University, November 1999.
- [36] Saggese, G.P.; Wang, N.J.; Kalbarczyk, Z.T.; Patel, S.J.; Iyer, R.K.; *An experimental study of soft errors in microprocessors*, Micro, IEEE Volume 25, Issue 6, Nov.-Dec. 2005 Page(s):30 - 39
- [37] Chad Scarbrough, Mohamed A. Gomaa, T.N. Vijaykumar, Irith Pomeranz; *Transient Fault Recovery for Chip Multiprocessors*, 30th Annual International Symposium on Computer Architecture (ISCA), June 2003.
- [38] Austin, T.M.; *DIVA: a reliable substrate for deep submicron microarchitecture design*, Microarchitecture, 1999. MICRO-32. Proceedings. 32nd Annual International Symposium on, 16-18 Nov. 1999 Page(s):196 - 207
- [39] Oh, N.; Shirvani, P.P.; McCluskey, E.J.; *Error detection by duplicated instructions in super-scalar processors*, Reliability, IEEE Transactions on, Volume 51, Issue 1, March 2002 Page(s):63 - 75
- [40] Oh, N.; Shirvani, P.P.; McCluskey, E.J.; *Control-flow checking by software signatures*, Reliability, IEEE Transactions on, Volume 51, Issue 1, March 2002 Page(s):111 - 122