2011

Analyzing and Predicting Processor Vulnerability to Soft Errors Using Statistical Techniques

Lide Duan
Louisiana State University and Agricultural and Mechanical College, lide.duan@gmail.com

ANALYZING AND PREDICTING PROCESSOR VULNERABILITY TO SOFT ERRORS USING STATISTICAL TECHNIQUES

A Dissertation

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

in

The Department of Electrical and Computer Engineering

by
Lide Duan
B.S., Shanghai Jiao Tong University, Shanghai, China 2006
December 2011

ACKNOWLEDGEMENTS

I would like to dedicate this dissertation to my parents, my wife, and my parents-in-law, for their continuous support and encouragement throughout my entire life.

This dissertation could not have been completed without the help and support from a lot of people that I am grateful to. First of all, I would like to thank my advisor, Dr. Lu Peng, for his guidance and suggestions during my Ph.D. study. All of the works presented in this dissertation came from constant support and discussions with Dr. Peng. I would also like to thank Dr. Bin Li from Department of Experimental Statistics for all the collaboration we had. Most of my works needed help from Dr. Li's expertise in statistics. Furthermore, I want to thank Dr. Doris Carver (my Minor Professor from Computer Science), Dr. Jagannathan Ramanujam and Dr. David Koppelman (the professors in my committee) for spending time supervising my dissertation and attending my defense.

I am thankful to the Department of Electrical and Computer Engineering for providing assistantship throughout my Ph.D. study. Also, I am very grateful to the Louisiana Optical Network Initiative (LONI) and the LSU Graduate School for providing LONI Graduate Fellowship and Dissertation Year Fellowship to support my last two year's Ph.D. research.

Finally, I would like thank all the friends I met at LSU for making my life here wonderful and memorable.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS ii

LIST OF TABLES v

LIST OF FIGURES vi

ABSTRACT ix

CHAPTER 1. INTRODUCTION 1
1.1 Soft Errors 1
1.2 Architectural Vulnerability Factor (AVF) 3
1.2.1 The Definition of AVF 3
1.2.2 The Importance of AVF 4
1.2.3 The Problem with AVF Measurement 5
1.3 Statistical Applications in Computer Architecture 6
1.4 AVF Modeling and Prediction 7
1.4.1 Dynamic AVF Prediction from Processor Performance Metrics 8
1.4.2 AVF Prediction on Multi-Threaded Processors with Resource Sharing 9
1.4.3 Universal Rules Guided Design Parameter Selection 9

CHAPTER 2. BACKGROUND 11
2.1 AVF Calculation 11
2.2 Boosted Regression Trees (BRT) 14
2.3 Patient Rule Induction Method (PRIM) 17

CHAPTER 3. DYNAMIC AVF PREDICTION FROM PERFORMANCE METRICS 20
3.1 Summary 20
3.2 Experimental Setup 22
3.3 Versatile AVF Prediction 24
3.3.1 Prediction within the Same Processor Configuration 24
3.3.2 Prediction across Different Processor Configurations 28
3.3.3 AVF Behavior Analysis and Model Interpretation 31
3.3.4 A Comparison between BRT and Linear Regression 34
3.4 Fast AVF Estimation 37
3.5 Case Study: PRIM-Based ROB Redundancy 38
3.5.1 The Approach 39
3.5.2 Performance Degradation Analysis 41
3.6 Related Work 43
3.7 Conclusions 45

CHAPTER 4. AVF PREDICTION ON MULTI-THREADED PROCESSORS 46
4.1 Summary 46
4.2 Two-Level Prediction Methodology 50
4.3 Impact of Multi-Programmed Resource Contention on AVF 54

4.3.1 Experimental Setup 54
4.3.2 Impact on SMT Private Structures 56
4.3.3 Impact on SMT Shared Structures 57
4.3.4 Impact on CMPs 59
4.4 Two-Level AVF Prediction for Multi-Programmed Workloads 60
4.4.1 Single-Threaded Universal Model Validation 60
4.4.2 Combing the Two Levels 62
4.4.3 Model Scalability 63
4.5 Extension to Multi-Threaded Workloads 65
4.6 Case Study: Soft Error Resilient Thread-to-Core Scheduling 67
4.7 Related Work 70
4.8 Conclusions 70

CHAPTER 5. UNIVERSAL RULES GUIDED DESIGN PARAMETER SELECTION 72
5.1 Summary 72
5.2 Design Parameter Selection for Uniprocessors 75
5.2.1 Experimental Setup 75
5.2.2 Application-Specific Design Parameter Selection 77
5.2.3 Universal Rules Guided Design Parameter Selection 80
5.2.4 Universal Rules Validation 83
5.3 Balancing Reliability, Performance and Power for Multiprocessors 85
5.3.1 Experimental Setup 86
5.3.2 Optimizing Individual Metrics 87
5.3.3 Balancing Multiple Metrics 90
5.4 Related Work 91
5.5 Conclusions 92

CHAPTER 6. SUMMARY AND FUTURE WORK 93
6.1 Summary 93
6.2 Future Work 95
6.2.1 Application Level Soft Error Propagation Analysis 95
6.2.2 Inter-Layer Coordinated Fault Tolerance 97
6.2.3 Constructing Reliable High Performance Computing (HPC) Systems 97

REFERENCES 99

APPENDIX A. PERMISSIONS TO USE COPYRIGHTED MATERIALS 103

APPENDIX B. AUTHOR'S PUBLICATIONS 106

VITA 108

LIST OF TABLES

Table 2-1. The ACE bits in different dynamic instructions 12

Table 3-1. The Alpha-21264-like machine configuration 23

Table 3-2. Workloads and SimPoints 24

Table 3-3. Explanation of variable names 24

Table 3-4. Configurations used in Section 3.3.2. The training set contains the 48 phase files of *cfg1* to *cfg12* (white), and the test set includes the 12 phase files of *cfg13* to *cfg15* (gray). 29

Table 4-1. Benchmarks used in multi-programmed workloads. The number of fast-forwarded instructions (unit: 100M) is shown after the benchmark name. 55

Table 4-2. Processor configuration design space composed of parameters P_1 to P_9. The values shown in bold are used in our baseline setting (Table 4-1). 61

Table 5-1. SPEC benchmarks used in the uniprocessor study. 76

Table 5-2. The uniprocessor design space is composed of parameters P_1 to P_8. Branch predictors are renamed to *BP1* to *BP8* for later use. 77

Table 5-3. Rules for optimizing individual structure's AVF for a set of benchmarks. "*Width/ALUs*" is the combination of processor width, # of integer ALUs, and # of FP ALUs; '&' refers to 'AND'; '||' refers to 'OR'. 78

Table 5-4. The multiprocessor design space is composed of parameters M_1 to M_9. Only multiprocessors with homogeneous cores are considered. The entire space size is 1,458,000 87

Table 5-5. Universal rule sets for optimizing different metrics for multiprocessors 88

LIST OF FIGURES

Figure 1-1. Strike changes state of a single bit. In this example, 1 will be flipped to 0. 2

Figure 2-1. The equation to calculate the AVF of a structure for a program phase.....13

Figure 2-2. BRT-based algorithm used in this dissertation.....15

Figure 2-3. Illustration of the BRT algorithm in two-dimensional input space.....16

Figure 2-4. PRIM training procedure, including peeling and pasting.....18

Figure 3-1. Input variable importance (within the same configuration) 25

Figure 3-2. Prediction results on different workloads (the 4 phase 1 on the left) and future phases (the 19 phase 2 on the right).....26

Figure 3-3. Prediction results in terms of relative error rates.....26

Figure 3-4. Empirical CDF on absolute errors in the with-configuration study 28

Figure 3-5. Input variable importance (across different configurations) 30

Figure 3-6. Prediction results on different configurations 30

Figure 3-7. Empirical CDF on absolute errors in the cross-configuration study 31

Figure 3-8. Measured and predicted IQ AVF curves for gcc.integrate and craft in the within-configuration study.....32

Figure 3-9. Partial dependence of the IQ AVF on the two most important variables in the within-configuration study.....33

Figure 3-10. Partial dependence of the ROB AVF on the most important variable in the within-configuration study.....34

Figure 3-11. R-squares on training and test sets over model size in linear regression and BRT methods (Left), and the scaled coefficients for the first ten selected variables in linear regression (Right). 35

Figure 3-12. R-square curves for the training and test sets with different learning rate *v* in BRT 36

Figure 3-13. Fast estimation of the ROB AVF in the within-configuration study.....38

Figure 3-14. Derived rules for the ROB AVF estimation 38

Figure 3-15. PRIM-Based ROB redundancy algorithm.....40

Figure 3-16. IPC variation for *gcc.scilab* in phase 2 of both schemes: without ROB redundancy and with PRIM-based ROB redundancy.....42

Figure 3-17. IPC variation for *bzip* in phase 2 of both schemes: without ROB redundancy and with PRIM-based ROB redundancy.....42

Figure 3-18. ROB occupant rate for *gcc.scilab* in phase 2 and *bzip2* in phase 2.....43

Figure 4-1. The AVF variations of gcc (SPEC 2000) when it is co-scheduled with different benchmarks on SMT/CMP.....47

Figure 4-2. Comparison between the proposed two-level scheme and prior one-level scheme 49

Figure 4-3. The first level universal model training using BRT. Only the inputs from a certain benchmark B_k are shown.....52

Figure 4-4. The second level universal model training using BRT 52

Figure 4-5. An overview of the two-level AVF prediction on a multi-threaded processor 53

Figure 4-6. Input variable importance of ROB AVF in SMT processors.....57

Figure 4-7. Input variable importance of Register File AVF in SMT processors.....58

Figure 4-8. Partial dependence plot of Register File AVF on the two most important input variables in SMT processors.....58

Figure 4-9. The partial dependence of CMP core AVF on its most important input.....59

Figure 4-10. Prediction accuracy (core AVF) of the first level model. This model predicts 100 configurations for each of the 30 benchmarks in the training set (left) and 400 configurations for each of the 8 benchmarks in the test set (right).....62

Figure 4-11. Prediction accuracy from two-level prediction 63

Figure 4-12. Prediction accuracy when scaling to 4 and 8 threads.....64

Figure 4-13. Prediction accuracy of SPLASH2 applications. The number at the end of an application name indicates the number of threads enabled.....67

Figure 4-14. Input variable importance of the predictor for multi-threaded applications.....67

Figure 4-15. An example of identifying the optimal thread-to-core scheduling on a 2-way 2-core CMT processor. The optimal scheduling is shown in solid lines.....68

Figure 4-16. The runtime AVF behavior of different schedules. The SOS curve indicates the online AVF variation when "Sample-Optimize-Symbios" job scheduler is utilized. 70

Figure 5-1. The AVF rank of a certain configuration varies significantly across different multi-threaded workloads. A lower rank indicates a lower AVF value that is favored in a reliable processor 73

Figure 5-2. ROB AVF of *mcf* (SPEC 2000) varies with different branch predictors 79

Figure 5-3. Register File AVF of *milc* (SPEC 2006) varies with different ROB sizes 79

Figure 5-4. Functional Unit AVF of *fma3d* (SPEC 2000) varies with different combinations of processor width and # of ALUs 80

Figure 5-5. ROB and LSQ AVFs of *applu* (SPEC 2000) vary with different LSQ sizes 81

Figure 5-6. Rule Set I (Optimizing Uniprocessor AVF) 82

Figure 5-7. Validation of Rule Set I on the test benchmarks 85

Figure 5-8. Validation of Rule Set II, III, IV on the test multi-threaded benchmarks. The number at the end of a benchmark's name indicates the number of threads. 89

Figure 5-9. The objective function used in balancing multiple metrics 90

Figure 6-1. An application is a directed graph of routines 96

ABSTRACT

The shrinking processor feature size, lower threshold voltage and increasing on-chip transistor density make current processors highly vulnerable to soft errors. *Architectural Vulnerability Factor (AVF)* reflects the probability that a raw soft error eventually causes a visible error in the program output, indicating the processor's susceptibility to soft errors at architectural level. The awareness of the AVF, both at the early design stage and during program runtime, is greatly useful for designing reliable processors. However, measuring the AVF is extremely costly, resulting in large overheads in hardware, computation, and power. The situation is further exacerbated in a multi-threaded processor environment where resource contention and data sharing exist among different threads. Consequently, predicting the AVF from other easily-measured metrics becomes extraordinarily attractive to computer designers.

We propose a series of AVF modeling and prediction works via using advanced statistical techniques. First, we utilize the *Boosted Regression Trees (BRT)* scheme to dynamically predict the AVF during program execution from a variety of performance metrics. This correlation is generalized to be across different workloads, program phases, and processor configurations on a single-threaded superscalar processor. Second, the AVF prediction is extended to multi-threaded processors where the inter-thread resource contention shows significant and non-uniform impacts on different programs; we propose a two-level predictive mechanism using BRT as building blocks to characterize the contention behavior. Finally, we employ a rule search strategy named *Patient Rule Induction Method (PRIM)* to explore a large processor design space at the early design stage. We are capable of generating selective rules on important configuration parameters. These rules quantify the design space subregion yielding lowest values of the response, thereby providing useful guidelines for designing reliable processors while achieving high performance.

CHAPTER 1. INTRODUCTION

This dissertation focuses on soft error reliability of current processors, including single-threaded and multi-threaded processors. It utilizes a number of machine learning and statistical techniques to perform the analysis and prediction. The three works presented in this dissertation (Chapter 3, 4, 5, respectively) build a strong foundation for future opportunities to improve processor robustness against soft errors via scientific approaches. Most of these achievements have been published in high quality computer architecture conferences and journals.

This chapter presents an informative introduction to these works. It starts from introducing background in soft errors and architectural vulnerability, the metric we used to quantify the soft error masking effect at computer architecture level. The high cost in measuring the soft error vulnerability motivates our series of works in predictions. This chapter finally briefly describes the three works performed in my dissertation research along with the reasoning behind them.

1.1 Soft Errors

Soft errors have become an important factor in degrading the reliability of current high performance processors. They occur mainly due to electronic noises caused by energetic nuclear particles in the environment. There are basically two types of these nuclear particles: one is the alpha-particles generated in the decay of radioactive atoms existing in many materials in our common lives; the other is the neutrons and pions generated from cosmic rays. As shown in Figure 1-1, these particles may invert the state of a logic device (from ‘0’ to ‘1’, or from ‘1’ to ‘0’) when the resulted charge has been accumulated to a sufficient amount. IBM experiments [55] demonstrated that a single alpha-particle could cause four memory cells to change their content

from a one to a zero in a 64Kb DRAM memory chip. This kind of errors is termed as soft errors or transient faults, because only the stored data is destroyed but the circuit itself is not damaged.

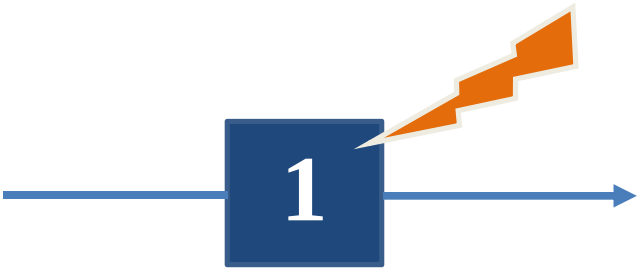

Figure 1-1. Strike changes state of a single bit. In this example, 1 will be flipped to 0.

If the corrupted bit is critical to the running program, the resulted error may propagate along with program execution to the output. In these cases, soft errors produce erroneous outputs that affect program correctness, significantly degrading the reliability. With the feature size and supply voltage scaling down to extremely small values (which effectively makes a bit more easily be flipped), current processors become highly vulnerable to soft errors. Many industry companies have reported the observation of soft errors causing severe damage on large servers in the past decades. For example, cosmic ray strikes on the L2 cache have caused Sun Microsystems's flagship servers to suddenly crash; also, error logs of large servers in many companies have documented various mysterious errors due to strikes.

Physical solutions to improve the reliability against soft errors are hard. For instance, the absorbent required to provide shielding is not practical (i.e. approximately > 10 feet of concrete); radiation-hardened cells come with significant penalty in performance, area, and cost. Nevertheless, these techniques may help alleviate the impact of soft errors at higher costs, but not completely remove it. In contrast, this dissertation analyzes and solves this problem from a different

angle: we propose a series of works at computer architecture level that can effectively and efficiently mitigate processor vulnerability to soft errors.

1.2 Architectural Vulnerability Factor (AVF)

To characterize a processor's soft error reliability, one should look at its effective soft error rate, i.e. the amount of actual errors resulted from raw soft errors in a time unit. Hence, the effective soft error rate (SER) is the product of raw SER and the probability that a soft error produces a visible error in the program output. The former is dependent on many factors at the circuit level including the critical charge of the circuits, processor area, temperature, etc.; while the latter is quantified at the architectural level by the Architectural Vulnerability Factor (AVF) [33][3]. In this dissertation, we focus on minimizing the AVF for soft error resilient designs at the architectural level. The term "reliability" used in this dissertation (as well as many prior publications) refers to the processor robustness to *raw* soft errors.

1.2.1 The Definition of AVF

A raw soft error does not necessarily affect the final output of a program. For example, a bit flip in an empty Reorder Buffer entry will not cause any error in the program execution; similarly, overwriting the incorrect bit in a register before it is used again prevents the propagation of the error. Based on this observation, a processor structure's Architectural Vulnerability Factor (AVF) [33] was defined as the probability that a raw soft error occurring in that structure finally produces a visible error in the program output. For example, the Branch Predictor's AVF is 0% since it doesn't matter at all for program correctness; on the other hand, the Program Counter (PC)'s AVF is almost 100% because almost all the bits in the PC always matter. A higher AVF

value indicates that the processor is more vulnerable to soft errors; the AVF characterizes the processor's vulnerability to soft errors at architectural level.

A common approach to calculate a processor structure's AVF is via Architecturally Correct Execution (ACE) analysis [33][3]: count the number of bits that are required for correct execution, and then divide it by the total number of bits of the structure. Using the ACE analysis method, many publications (e.g. [33][19]) have reported a large masking effect of raw soft errors at architectural level. That said, a key processor structure usually shows an AVF below 40%, but with a large variation over time.

1.2.2 The Importance of AVF

The AVF provides computer architects with an indicator of the system's susceptibility to soft errors. The awareness of the AVF, both at the early design stage and during program runtime, is greatly useful for cost-effective reliable processor designs. One example would be some form of partial redundancy described below.

Researchers [35][46][32][20] proposed Redundant Multithreading (RMT) to detect/recover from soft errors. The basic idea is to run two copies of the same program as separate threads with identical inputs on a Simultaneous Multithreading (SMT) or a Chip Multicore (CMP) processor. By comparing the outputs of the two threads, we can detect an effective soft error. A big drawback of RMT is the significant performance degradation due to the resource contention between the two threads. Mukherjee et al. [32] reported a 32% performance degradation of running the two threads simultaneously compared to only running a single copy of the program. In such cases, the AVF can serve as an indicator to turn on/off the redundant thread: if the AVF of current program phase is very low (which means an error is not likely to happen), we can temporarily

disable the redundant thread to boost the performance by eliminating the inter-thread contention. Therefore, dynamically tracking the AVF during program execution is greatly useful for achieving a good tradeoff between system performance and soft error reliability.

1.2.3 The Problem with AVF Measurement

The AVF provides useful guidelines in designing reliable processors, but its measurement is extremely expensive in terms of hardware and computation. To measure the AVF, one can use *Statistical Fault Injection* [49][29] or *Architecturally Correct Execution (ACE)* analysis [33]. The former requires a large number of experiments that randomly inject errors into program execution; while the latter needs to implement a post-commit analysis window to identify the hardware bits that are required for correct execution. Regardless, either of these two methods results in costly overhead and significant performance degradation.

Specifically in our work, we followed the ACE method to calculate the AVF in our simulators. The implemented post-commit analysis window [33][18] contains the most recent 40K committed instructions. The processor inserts the instructions into this window after they are committed, and also maintains the dependencies among them. At the other end of this window, we can determine an instruction's type (e.g. a dynamically dead instruction, or a NOP instruction, etc), and use this information backward to calculate the AVF. Apparently, implementing this approach in the real processor requires additional hardware structures; additionally, the simulation in a simulator capable of measuring the AVF is significantly slowed down due to the additional computation incurred by the AVF measurement.

Furthermore, the ACE analysis can be also applied to multi-threaded processors' AVF measurement. However, it is even more involved than in single-threaded processors because the

instruction streams from different threads must be traced independently and thus accompanied with separate dependency chains. For multi-threaded workloads with data sharing, a system-wide, much bigger analysis window must be implemented. Therefore, an accurate AVF prediction in place of actual AVF measurement can save a great amount of hardware overhead, computation, power consumption, etc, thereby being very attractive to computer architects designing soft error resilient systems.

1.3 Statistical Applications in Computer Architecture

In recent years, there is a tendency to apply statistical and machine learning techniques to computer architecture research. The fundamental reason is because of the fast increasing number of design options. Different programs, and even different parts of the same program, may have distinct behaviors that interact with the hardware in different ways; on the other hand, the number of representative programs (i.e. benchmarks) and the number of possible hardware design choices are increasing geometrically. Consequently, it's critical that computer architects have efficient mechanisms by which they can estimate the overall impact of various designs on the system. Statistical and machine learning techniques have been chosen for this purpose.

One example would be the online AVF prediction. As explained in Section 1.2.3, dynamically predicting the AVF during program runtime eliminates the overhead in AVF measurement. Fu et al. [19] observed a fuzzy correlation between the AVF and a few common performance metrics. Walcott et al. [47] extended the input metrics set and used linear regression to reexamine this correlation. They performed a very accurate prediction, proving the existence of the correlation between the AVF and various processor performance metrics. Alternatively, Li et al. [28] developed an online algorithm to estimate processor structures' vulnerability using a modi-

fied error injection and propagation scheme [29][48]. The first two works in this dissertation, which will be described in the following sections, further generalized the correlation between AVF and performance metrics to be across workloads, execution phases and configurations. Regardless, certain predictive techniques have been used in these works to perform a fast and accurate prediction for metrics that are difficult to be directly measured.

Design space exploration is another kind of prediction that explores a huge design space and predicts the response for any design point in the space. Many researchers have done different variations of design space exploration. Ipek et al. [22] predicted performance of memory hierarchy, CPU and CMP design spaces using Artificial Neural Networks (ANNs); Similarly, Lee et al. [23] proposed to use spline-based regression to predict performance and power from a large design space. It's also possible to derive optimal points based on their predictive models via exhaustive prediction in Pareto Analysis [24]. There are also a number of studies discussing design space exploration on performance and/or power [30][31]. For the correlation between the AVF and configuration parameters, Cho et al. [8] predicted the dynamics of power, CPI and the AVF using a combination of wavelets and neural networks. They also followed the same approach to predict the average soft error vulnerability and its tradeoff with performance [9]. The third work presented in this dissertation follows this approach but derives universal guidelines via exploring the design space that can be applied across programs.

1.4 AVF Modeling and Prediction

The fundamental idea of this dissertation is to utilize advanced statistical techniques to model and predict the AVF from other easily-measured metrics, including performance metrics (e.g. structure occupancy rates), configuration parameters (e.g. structure sizes), and so on. Upon

accurate predictions, we are able to estimate the AVF values for a processor without AVF measurement mechanism, thereby eliminating the overheads in measuring the AVF. Our first work [11][27] predicts the instantaneous AVF from a large set of processor performance metrics on a single-threaded superscalar processor. This work actually bridges the gap between the difficult AVF measurement and easily-obtained performance measurements. Second, the AVF prediction [13] is performed on multi-threaded processors, e.g. Simultaneous Multithreading (SMT) or Chip-Multiprocessors (CMP), where the inter-thread resource contention shows significant impacts on the AVF. A two-level predictive mechanism is proposed to correlate the AVF with underline configuration and contention behaviors in shared structures. Finally, we also use a rule search strategy to identify the designs that have lowest AVF values from a large design space consisting of key configuration parameters [14][12]. The generated selective rules provide computer architects with useful guidelines in designing reliable processors while achieving high performance at the pre-silicon stage.

The main contributions of this dissertation are summarized as follows:

1.4.1 Dynamic AVF Prediction from Processor Performance Metrics

By utilizing a nonparametric tree-based predictive modeling scheme named Boosted Regression Trees (BRT), we are capable of predicting the AVF from a variety of performance metrics. More specifically, we trained a predictor using various performance metrics and the measured AVF from a large number of program phases as the input and output, respectively. The predictor is then tested on other program phases to demonstrate its accuracy. This prediction turns out to be valid across workloads, execution phases, and underline configurations. We also quantitatively demonstrate the input variable importance to the AVF and how the AVF is affected by the most important inputs. In addition to accurate predictions, another statistical technique (i.e.

Patient Rule Induction Method or PRIM) is employed to summarize a few simple but useful selective rules on the performance measurements. Applying these rules online can efficiently identify the program intervals vulnerable to soft errors.

1.4.2 AVF Prediction on Multi-Threaded Processors with Resource Sharing

The inter-thread resource contention and sharing significantly and non-uniformly affect the AVF of different threads running simultaneously on a multi-threaded processor. We propose a two-level predictive mechanism to predict the AVF under contention. At the first level, a unified model predicts a certain program's AVF when it runs alone on a single-threaded processor. The output of the first level model, along with a few key structures' occupancy rates measured when the program runs against other program(s), are the inputs to the second level model, which finally predicts the program's AVF under contention. Consequently, we can obtain an accurate AVF estimation for a program co-scheduled with different programs on an SMT/CMP with any configuration in the design space. In practice, the proposed scheme can be used to find soft error resilient thread-to-core mapping for both homogeneous and heterogeneous multi-threaded processors.

1.4.3 Universal Rules Guided Design Parameter Selection

We propose an effective approach to identify the configurations that have consistently low AVF values from a huge design space. Those identified configurations are inherently reliable to soft errors. Specifically, we characterize the design space using *Patient Rule Induction Method* (PRIM) to generate a set of selective rules on key design parameters. Applying these rules on the design space effectively identifies the design space subregion within which the output variable is considerably smaller (i.e. "valley seeking") than its average value over the entire design space.

Therefore, the design configurations selected by the generated rules are inherently resilient to potential soft errors. This technique provides computer architects with useful guidelines to design reliable processors at pre-silicon stage. Furthermore, the proposed method is performed on multiprocessors to simultaneously balance multiple design metrics and achieve a proper tradeoff among reliability, performance, and power.

CHAPTER 2. BACKGROUND

2.1 AVF Calculation

The concept of Architectural Vulnerability Factor (AVF) was originally proposed by Mukherjee et al. [33]; Biswas et al. [3] extended it to address-based structures. There are two approaches to measure the AVF: *Statistical Fault Injection (SFI)* [49][29] and *Architecturally Correct Execution (ACE)* analysis [33]. For SFI, a microarchitectural fault injector is needed for the simulator; a bit is randomly chosen from the microarchitectural state to be flipped at a random time during program execution; the microarchitectural state and simulation results are compared with the correct execution (i.e. the golden run) when simulation finishes. Therefore, calculating the AVF requires a large number of such fault injection experiments. The other approach, which we implemented in our simulators, is via ACE analysis; it provides a tight upper bound [4] on the soft error vulnerability of various processor structures. A unified framework named Sim-SODA [18] to study the AVF behavior in a superscalar processor has been released. We generally follow the ACE approach described in these papers to calculate the AVF in our simulators.

In principle, we need to identify which bits in a dynamic instruction affect the correct execution of the program. These bits are called ACE bits since a fault in any of these bits will cause a visible error in the final output. On the other hand, we classify the dynamic instructions into five categories: NOP instructions which do not change any processor state; prefetch instructions which bring data into cache in advance but do not change processor states as well; dynamically dead instructions whose computed results are not used by subsequent instructions; unknown instructions whose types cannot be determined due to insufficient information; the remaining are

ACE instructions performing useful computation. Table 2-1 summarizes the ACE bits for each of these instruction types.

Table 2-1. The ACE bits in different dynamic instructions

Dynamic Instruction Type	ACE Bits
NOP, prefetch instructions	*opcode* field
dynamically dead instructions	*opcode* field, destination register field
unknown instructions, ACE instructions	all bits (except masked source registers)

For NOP and prefetch instructions, as long as the processor correctly decodes them via the *opcode* field, the execution correctness would not be violated (although a different data block may be prefetched due to an error in other bits). For dynamically dead instructions, besides *opcode* field we need to guarantee that the result is written into the correct register. Some instructions are unknown of their types since they are near the end of the simulation and we don't have sufficient information to determine their types (e.g. whether the result computed by such an instruction will be used in the future). To have a conservative estimate of the AVF, we consider unknown instructions to have all bits as ACE. Finally, an ACE instruction usually has its all bits to be ACE with some exceptions in the masked source register field. For instance, if an "AND" instruction has one source register equal to zero, the other source register is actually masked (since the result is always zero regardless of the value in this register) and a soft error is tolerant in it.

Furthermore, to calculate the AVF for a processor structure (e.g. ROB), we also need to know how long each instruction stays in the structure (via recording the cycle numbers the instruction gets in/out of this structure). We then use the equation [33] in Figure 2-1 to calculate the AVF of a processor structure for a certain phase of program in execution:

$$\frac{\sum \text{residency (in cycles) of all ACE bits in the structure for the program phase}}{\text{Total number of bits in the structure} * \text{total execution cycles for the program phase}}$$

Figure 2-1. The equation to calculate the AVF of a structure for a program phase

The main difficulty in the above approach to calculate the AVF resides in determining the exact type of each dynamic instruction. For example, to determine whether an instruction (say *inst*) is dynamically dead, we have to track a large amount of instructions that follow *inst*. Among those subsequent instructions, if the destination register of *inst* is not used at all, *inst* is first-level dynamically dead; if the destination register is only read by the instructions turn out to be dynamically dead as well, *inst* is transitively dynamically dead; otherwise, *inst* is an ACE instruction. Hence, this determination involves a lot of computation, thus significantly slowing down the simulation. In our simulator, we implement a post-commit analysis window [33][18], which tracks the most recent 40K committed instructions, to determine the instruction type after the instruction goes through the entire window. The dependencies among the instructions have to be maintained in the window.

In summary, the AVF measurement suffers from two main disadvantages. First, it results in large overhead in computation, power, hardware, etc. This dissertation is motivated from this disadvantage. Second, a gap exists between the performance simulation and the AVF measurement due to the post-commit analysis window. In other words, the AVF measurement of a certain program interval is delayed 40K instructions (the length of the window). Our work in Chapter 3 bridges the gap via AVF prediction from online performance measurements.

2.2 Boosted Regression Trees (BRT)

Boosted Regression Trees (BRT), originally proposed by Friedman [16], is an ensemble technique that aims to improve the performance of a single model via fitting and combining many models for prediction. BRT employs two algorithms: "regression trees" from Classification and Regression Trees [5] (CART), and "boosting" which builds and combines a collection of tree models. CART is a binary recursive partitioning algorithm. Initially, it splits the input space into two regions, and models the response by a constant for each region. For each of these regions, an input dimension and a split point are chosen to achieve the best fit. This can be visualized as a parent node splitting into two child nodes. This procedure is recursively applied until the constructed tree reaches a certain depth. On the other hand, boosting is one of the recent enhancements to tree-based methods. In boosting, models such as regression trees are fitted iteratively to the training data, and an appropriate method is used to gradually increase emphasis on observations modeled poorly by the existing collection of trees.

The detailed BRT algorithm used in this dissertation is described in Figure 2-2. We consider a problem with n observations $\{y_i, \mathbf{x}_i\}$, $i=1,2,\ldots,n$, where $\mathbf{x}_i$ is a p-dimensional input vector and y_i is the response. In Step 2(b), a binary regression tree is constructed based on the distribution of current residuals. At each node of the tree, the algorithm partitions current region in a way that the resulted new prediction function produces the minimal Mean Square Error (MSE) among all possible split points. The tree construction terminates when its depth reaches a certain number. In this work, we set the maximum tree depth to be 3, indicating that H is not greater than 8. In Step 2(d), $I(\bullet)$ is an indicator function which returns 1 (otherwise 0) if its argument is satisfied. ν is a parameter between 0 and 1, controlling the learning rate of the procedure. Empirical results have shown that smaller values of ν always lead to better generalization errors [16]. In this

study, we fix ν at 0.01. Figure 2-3 illustrates the BRT algorithm in a two-dimensional input space. We can see that M regression trees are constructed corresponding to the M iterations of the algorithm. This figure shows the first one (Tree 1) as an example in which the input space is partitioned into five regions. Note that each tree is constructed based on the training data and the prediction function in current iteration, so they may be different from each other.

BRT is inherently nonparametric and can naturally handle mixed-type of input variables. Different from other parametric models, BRT does not make any assumption about the distribution of the values of the input variables, thus avoiding the transformations performed in preprocessing training data. BRT is also capable of capturing complex behaviors with a relatively small number of inputs. This is in contrast to some other multivariate nonlinear modeling techniques, in which extensive inputs from the analyst, analysis of interim results, and subsequent modifications of the method are required. Besides, BRT is insensitive to outliers, and unaffected by monotone transformations and different scales of input measurements.

1. Initialize $\hat{f}_0(\mathbf{x}_i) = \bar{y}$, where $\bar{y}$ is the average for $\{y_i\}$.
2. Repeat for $m = 1,2,...,M$:
   (a) Compute the current residuals: $r_{im} = y_i - \hat{f}_{m-1}(\mathbf{x}_i),\ i = 1,...,n.$
   (b) Partition the input space into H disjoint regions $\{R_{hm}\}_{h=1}^{H}$ based on $\{r_{im}, \mathbf{x}_i\}_{i=1}^{n}$.
   (c) For each region, compute the constant fit: $\gamma_{hm} = \arg\min_{\gamma} \sum_{\mathbf{x}_i \in R_{hm}} (r_{im} - \gamma)^2$
   (d) Update the fitted model: $\hat{f}_m(\mathbf{x}) = \hat{f}_{m-1}(\mathbf{x}) + \nu \times \gamma_{hm} I(\mathbf{x} \in R_{hm})$
3. End algorithm.

Figure 2-2. BRT-based algorithm used in this dissertation.

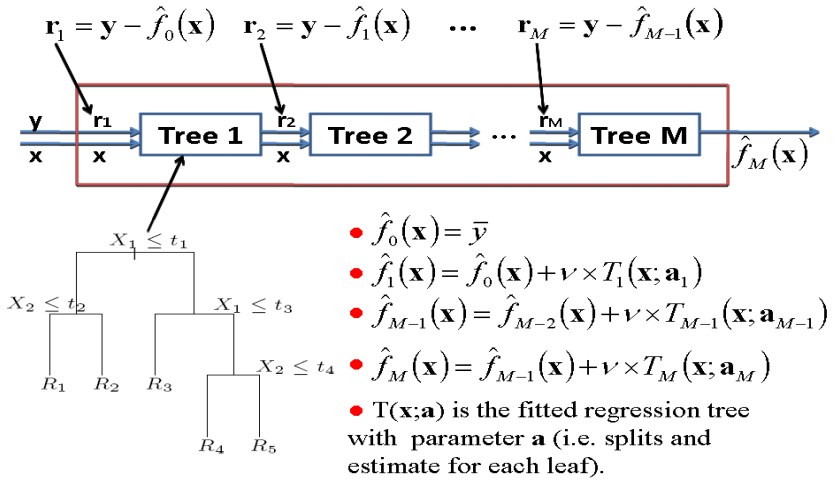

Figure 2-3. Illustration of the BRT algorithm in two-dimensional input space.

In addition to accurate prediction, BRT also provides visualized model interpretations including input variable importance to the response and the partial dependence of the response on the most important variables. The relative importance of a variable is measured by accumulating the number of times the variable (dimension) is selected for splitting a region during the M iterations. Each increment to this number is weighted by the improvement in the MSE as a result of the corresponding split. The relative importance is scaled so that the sum adds to 100%, with a higher number indicating a stronger influence on the response. On the other hand, a partial dependence plot shows the effect of a subset of input variables on the response after accounting for the average effect of all other input variables in the model. Given any subset $\mathbf{x}_\mathbf{s}$ of the input variables indexed by $\mathbf{s} \subset \{1, \cdots, p\}$, the partial dependence of $f(\mathbf{x})$ is defined as $F_s(\mathbf{x}_\mathbf{s}) = E_{\mathbf{x}_{\setminus \mathbf{s}}}[f(\mathbf{x})]$, where $E_{\mathbf{x}_{\setminus \mathbf{s}}}[\cdot]$ refers to the expectation over the joint distribution of all the input variables with indices not in $\mathbf{s}$. In practice, partial dependence can be estimated from the training data by $\hat{F}_\mathbf{s}(\mathbf{x}_\mathbf{s}) = (1/n)\sum_{i=1}^{n} \hat{f}(\mathbf{x}_\mathbf{s}, \mathbf{x}_{i\setminus \mathbf{s}})$, where $\{\mathbf{x}_{i\setminus \mathbf{s}}\}_1^n$ are the data values of $\mathbf{x}_{\setminus \mathbf{s}}$.

2.3 Patient Rule Induction Method (PRIM)

The objective of PRIM [17] is to find a subregion in the input space that gives relatively low values for the output response. The identified input space subregion (or "box") is described as a set of simple "selective rules" in a form of $B = \bigcap_{j=1}^{p} \left(x_j \in s_j \right)$. x_j represents the j[th] input variable, and s_j is a subset of all possible values for the j[th] variable. Hence, the identified subregion B is the intersection of p subsets, each being from one input variable.

The box construction of PRIM consists of two phases: patient successive top-down peeling and bottom-up recursive pasting. Figure 2-4 visualizes this procedure. The top-down peeling starts from the entire design space. At each iteration, we have the following operations: a small subbox b within the current box B is removed; we calculate the output mean for the elements remaining in $B - b = \{x \mid x \in B \,\&\, x \notin b\}$, and try this operation in each dimension (i.e. try removing a different subbox from each input variable); we then choose the one that yields the smallest output mean value for the next box B-b. The above procedure is iteratively applied until the support of the current box B is below a chosen threshold β, which is the proportion of the design points remaining in the identified area. Note that for a categorical variable, an eligible subbox b contains only one element of the possible values of the variable in the current box B.

The pasting algorithm is simply the inverse of the peeling procedure. The reason for pasting is that at each iteration of peeling we only look one step ahead. The box boundary is thereby determined without knowledge of later iterations. Consequently, we may peel too much from the input space, and the final box can sometimes be improved by readjusting its boundaries. From the peeling result, the current box B is iteratively enlarged by pasting onto it a small subbox that minimizes the output mean in the new larger box. The subbox being pasted is chosen in the same

manner as in peeling. The bottom-up pasting is iteratively applied, successively enlarging the current box until the addition of the next subbox causes the output mean start increasing.

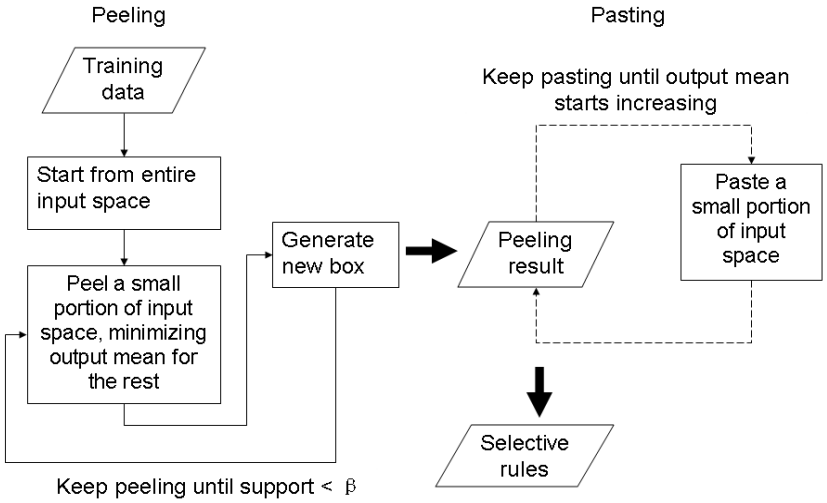

Figure 2-4. PRIM training procedure, including peeling and pasting.

Regarding the complexity of PRIM, the first peel requires at most $n*p$ operations, where n is the number of observations and p is the number of possible values of all input variables. The number of operations for each peel will decrease since fewer and fewer samples are left during peeling. On the other hand, PRIM performs approximately $-log(n)/log(1-\alpha)$ peeling steps, where α is the portion that is peeled off at each peeling iteration. An advantage of PRIM over greedy methods such as tree-based methods is its patience. For example, a binary tree partitions the data quickly because of its binary splits, while in PRIM each time only a small proportion (α) of data is peeled off. Hence, the solution of PRIM (hyper-boxes) is usually much more stable than the tree models. In other words, if the data are slightly changed, a tree structure may change dramatically but the PRIM solution is less affected. Moreover, if the optimal subspace is not connected, PRIM can generate a sequence of hyper-boxes instead of just one. Namely, after getting the first hyper-box, the PRIM procedure can be repeated on the remaining dataset. As a result, the disconnected subspace can also be covered. However, we found that in practice the leading one hy-

per-box usually covers most of the points with the smallest response values. Therefore, we only apply PRIM once to identify the desired subregion in the following sections.

Finally, all the description in this subsection assumes that the subregion that has the smallest response values is being extracted (i.e. “Valley Seeking”). However, we can certainly invert all the conditions above and look for the subregion that has the largest response values (i.e. “Bump Hunting”). The choice between these two depends on the problem being solved. In Chapter 3, we will use “Bump Hunting” to identify the program intervals showing highest AVF; in Chapter 5, we will use “Valley Seeking” to identify the designs that have the lowest soft error vulnerability.

CHAPTER 3. DYNAMIC AVF PREDICTION FROM PERFORMANCE METRICS[1]

3.1 Summary

The first work in this dissertation predicts the instantaneous AVF from processor performance metrics during program runtime. This work bridges the gap between the difficult AVF measurement and easily-obtained performance measurements. By utilizing the dynamic AVF prediction, we can eliminate the overhead due to measuring AVF, and also customize reliability enhancement mechanism to mitigate the performance degradation.

Several prior publications studied online AVF estimation. By observing a fuzzy correlation between the hardware AVF and some common performance metrics such as IPC, branch misprediction rate, cache miss rate, etc, Fu et al. [19] concluded that a simple performance metric was not a good indicator to the program soft error vulnerability behavior. Walcott et al. [47] reexamined the correlation by extending the variable set to 160 easily-measured time-varying processor metrics. They adopted a multivariate regression-based statistical model using 22 workloads as a training set to extract a quantitative relationship between the AVF and a small subset of the variables, and then applied the obtained predictor to another 4 workloads. By demonstrating a very accurate prediction of the AVF behavior, their work convincingly proved the existence of a correlation between the AVF and various processor performance metrics. However, they restricted their model training/test within one configuration, and only focused on the first SimPoints [38]

[1] © 2010 IEEE. Reprinted, with permission, from: B. Li, L. Duan, and L. Peng, "Efficient Microarchitectural Vulnerabilities Prediction Using Boosted Regression Trees and Patient Rule Inductions," IEEE Transactions on Computers (TC) – Special Issue on System Level Design of Reliable Architectures, vol. 59(5), pp. 593-607, May 2010.

of SPEC CPU 2000 benchmark suite. It is not clear that the predictor obtained from one set of phases (i.e. the first SimPoints) will give accurate estimation for another set of phases (e.g. the second SimPoints); also, the model developed under one configuration most likely would not work for other configurations.

In this chapter, we propose a *versatile* method which accurately predicts the AVF across different workloads, execution phases and processor configurations. Initially, a statistical model is trained using the first SimPoints measured from a set of workloads under a BRT-based algorithm; the trained model is then tested with other workloads that are not included in the training set. The testing results show that the prediction is very accurate. Within the same configuration, the trained model is also capable of predicting the vulnerabilities of the second SimPoints of all workloads. We then extend our model by adding the configuration parameters into the training variable set, and demonstrate a very high accuracy in predicting the AVF variations under different configurations.

Finally, to make our method easier to be used in practice, we propose a *fast* estimation approach which utilizes Patient Rule Induction Method (PRIM) to extract some simple selective "IF-ELSE" rules on important performance metrics. These rules can be used to monitor the performance variables during the program execution, and efficiently identify vulnerable intervals experiencing high AVF values.

The main contributions of this chapter are:

- **Versatile AVF Prediction**. Our proposed method accurately predicts the AVF across different workloads, execution phases, and processor configurations.

- **Model Interpretation and Comparison**. The proposed model can quantify the input variables' importance and the output response's dependence on the inputs. Our model is also much more stable than traditional linear regression.
- **Fast AVF Estimation**. The selective rules generated from the PRIM model significantly reduce the prediction complexity. This enables computer architects to efficiently identify highly vulnerable program intervals during runtime.
- **A Case Study of PRIM-Based ROB Redundancy**. It effectively reduces the ROB AVF to a very low level with negligible performance degradation.

3.2 Experimental Setup

We use *Sim-SODA* [18], a unified simulation framework that models soft error reliability of different microarchitecture structures in a microprocessor system, to measure the AVFs and a large set of performance metrics. *Sim-SODA* was developed based on *Sim-alpha* [10] which has been validated as an accurate Alpha 21264 simulator; it also has been incorporated with microarchitectural level AVF calculation methods for key processor structures. In this work, we use *Sim-SODA* to dump the time-varying AVF values for Integer Issue Queue (IQ) and Reorder Buffer (ROB). These two structures produce significant impacts on the processor vulnerability, so without losing generality our methods can be also used for other processor components.

Table 3-1 shows the Alpha-21264-like baseline machine configuration used in this work. In later sections, several key parameters will be tuned to generate 15 different configurations. For the experiments, all the integer benchmarks except one from the SPEC CPU 2000 suite are evaluated. The only exception is *gzip* whose simulation cannot be finished in a reasonable time in *Sim-SODA*. The floating point benchmarks of SPEC 2000 suite are not included in our experi-

ments because *Sim-alpha* cannot accurately model Alpha 21264 floating point pipeline (thus *Sim-SODA* does not support AVF measurements for FP workloads). In order to perform sufficient model training/test, we provide each benchmark with different inputs if possible; the total 19 workloads are listed in Table 3-2 in which the training set includes the white columns and the test set consists of the gray columns. Note that the training and test sets are disjoint.

Each workload is run for two 100-Million Instruction SimPoints [38]. Table 3-2 gives the number of instructions (unit: 100M) fast-forwarded to reach the SimPoints that we are interested in. In this work, we term each SimPoint (i.e. the execution of 100M instructions) as a "phase", and each 500K instructions within a SimPoint as an "interval". In other words, for each workload, we simulate two phases, each containing 200 intervals. The granularity of dumping the AVFs and performance metrics is "interval", that is, the system records the AVF values (of IQ and ROB) and the values of 217 performance variables after the execution of every interval. We don't list all of them here because of the space, but the following subsections will analyze the most important ones. Table 3-3 explains the abbreviation of variable names.

Table 3-1. The Alpha-21264-like machine configuration

Pipeline stages	8
Fetch/slot/map/issue/commit width	4/4/4/4/11
Fetch/slot queue size	4/4
Issue queue size	20
Reorder buffer size	80
Load/store queue size	32/32
Integer register file size	41 (1-cycle read latency)
Integer ALUs/multipliers	4/4 (latency: 1/7)
Branch predictor	Hybrid (local: 1K+1K; global: 4K; choice: 4K)
L1 I/D cache	64KB (64B block, 2-way, access latency: 1/3)
L2 cache	2MB (64B block, 1-way, 7-cycle latency)
ITLB/DTLB	Each: 128 entries, fully-associative
Victim buffer	8 entries, 1-cycle latency

Table 3-2. Workloads and SimPoints

Benchmark	Phase 1	Phase 2	Benchmark	Phase 1	Phase 2	Benchmark	Phase 1	Phase 2
bzip2.source	4	104	*mcf*	1	37	*crafty*	114	252
eon.cook	78	187	*parser*	173	309	*gcc.200*	101	137
eon.kajiya	389	410	*perlbmk.makerand*	0	5	*gcc.integrate*	1	11
eon.rushmeier	210	213	*twolf*	2	122	*vortex.lendian3*	97	311
gap	83	239	*vortex.lendian1*	78	127			
gcc.166	0	20	*vortex.lendian2*	164	422			
gcc.expr	8	24	*vpr.route*	2	265			
gcc.scilab	38	112						

Table 3-3. Explanation of variable names

Abbreviation	Example	Meaning
xxx_count	load_q_writes_count	# writes to load queue in current interval
xxx_cumulative_count	ready_q_cumulative_count	the cumulative # ready queue entries in all cycles of current interval
xxx_average_count	rob_average_count	rob_cumulative_count / # cycles of current interval
xxx_cumulative_latency	fu_cumulative_latency	the cumulative # cycles that the committed instructions of current interval stayed in functional unit
xxx_occupant_rate	issue_q_occupant_rate	issue_q_average_count / issue_q_size

3.3 Versatile AVF Prediction

Generally, we believe that the AVF value of a key processor structure is a complex function of a large set of processor performance metrics. The exact form of the function may vary in different execution stages or different configurations. Nevertheless, our proposed BRT method is capable of identifying important features from a large set of performance variables and accurately predicting the vulnerabilities across workloads, execution phases, and different configurations. We show the AVF prediction in this section.

3.3.1 Prediction within the Same Processor Configuration

This subsection discusses the model training and test under our baseline setting (Table 3-1) to demonstrate that BRT accurately predicts the vulnerabilities of other workloads and future execution phases. Specifically, 15 phase files (workloads in the white columns in Table 3-2) are

used to train a BRT model, which is then applied to other 4+19 phase files (phase 1 of 4 workloads and phase 2 of all workloads, as shown in the gray columns in Table 3-2).

We first apply the BRT algorithm using all 217 performance variables. The 10 most influential variables are listed in Figure 3-1. One can refer to Section 2.2 for details of the algorithm and how to calculate the input variable importance. As can be seen, the number of valid entries (*cumulative_count*, *average_count*) and the cumulative latency that the committed instructions spent in the structure significantly contribute to the vulnerability of the structure. In addition, states of some other microarchitectural components (e.g. Ready Queue, Load/Store Queue, Register File, etc) also strongly affect the AVFs of IQ and ROB.

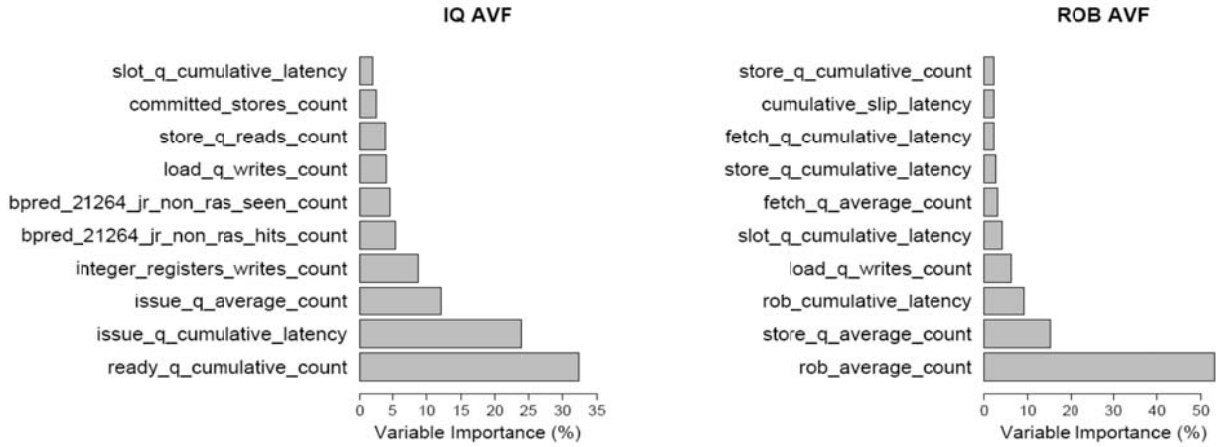

Figure 3-1. Input variable importance (within the same configuration)

After identifying the 10 most important performance metrics, we refit the BRT model by only using the 10 selected variables. This effectively reduces the number of dimensions of the input space from 217 to only 10, thus significantly reducing the model complexity. The prediction results of the workloads in the test set are shown in Figure 3-2. Note that in this figure the AVF values are shown in the range of 0 to 100 (rather than 0 to 1). For the 4 workloads (phase 1) on the left, the mean absolute errors (MAEs) for IQ and ROB AVFs are 0.93 and 0.55, respec-

tively, validating the ability of our model to accurately predict the AVF variation on different workloads. Furthermore, the MAEs for the second phases of all 19 workloads are almost all below 4 with only two exceptions *mcf* and *vpr* whose IQ errors reach about 8. The small average MAEs (2.23 for IQ and 1.16 for ROB) of the phase 2 files indicate that the cross-phase correlation between the vulnerability and performance metrics can be captured by our model.

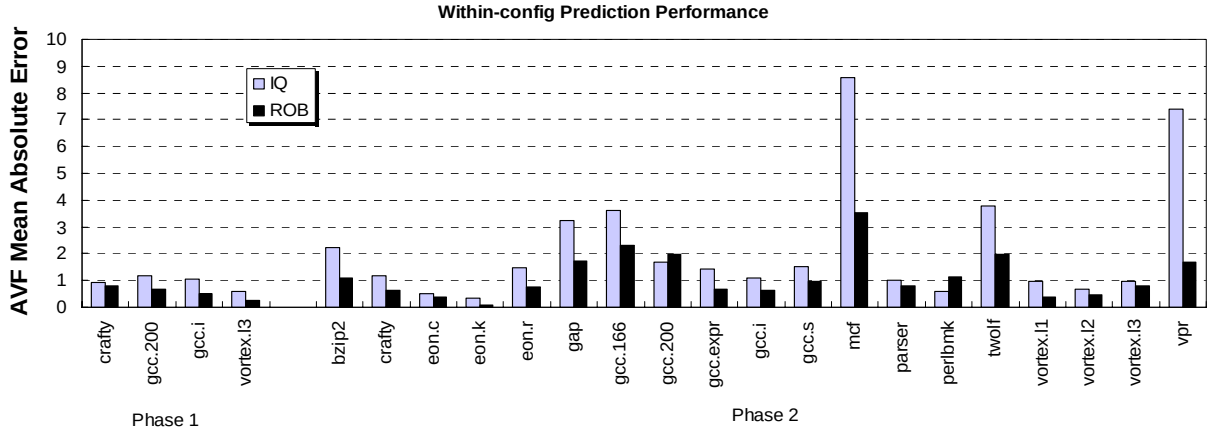

Figure 3-2. Prediction results on different workloads (the 4 phase 1 on the left) and future phases (the 19 phase 2 on the right)

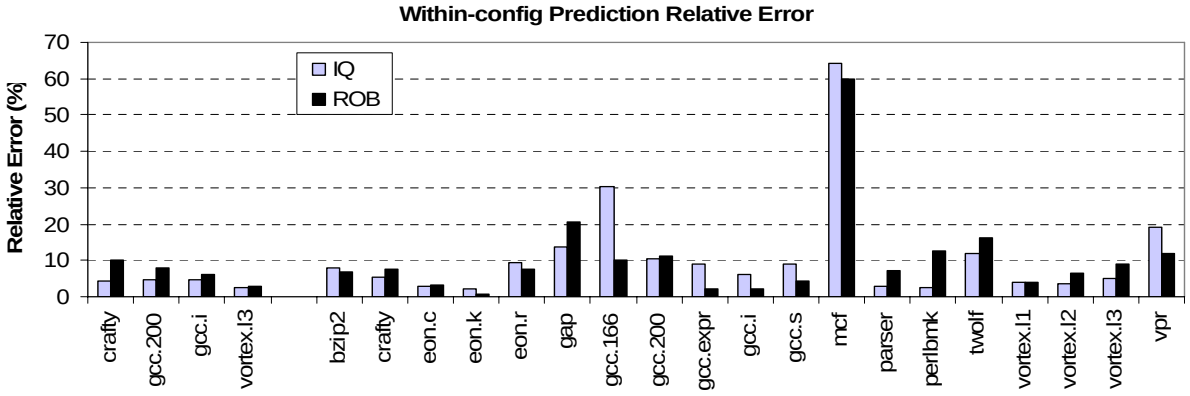

Figure 3-3. Prediction results in terms of relative error rates

For comparison purpose, we also present the relative error for each phase in Figure 3-3. As can be seen, most workloads in the test set are predicted with a relative error lower than 10%. Only *mcf* is an outlier that suffers from significant relative errors. From the simulation result, we found that its L1 data cache miss rate is only 0.0095 in the first SimPoint while this miss rate is 0.2877 in the second SimPoint. Therefore, the training data from *mcf* does not represent its typical memory-intensive behavior. We believe this is the reason why *mcf* shows high relative errors in the prediction.

Generally, we believe that relative error is not a good metric to report the prediction performance in this work. If we intended to look at the average AVF value throughout the entire phase (instead of a variation consisting of many intervals), relative error might be a good choice. However, our work addresses the instantaneous AVF curve which consists of hundreds of measured points within a phase, and some of them are very close to zero. The relative error could reach a very high value with a small absolute error. For example, if the true AVF measure is 1 and the predicted AVF is 2, the relative error is 100% though the absolute error is only 1. This kind of "outliers", though only a few, strongly affects the average of relative error, but does not reasonably reflect the prediction power of the model. Therefore, we will mainly focus on analyzing the mean absolute error of the AVF in this work.

Empirical Cumulative Density Function (CDF) is another way to report the prediction performance. From Figure 3-4, we see that over 90% of the intervals are predicted below absolute errors of 4.5 and 2.2 for the IQ and ROB AVFs, respectively.

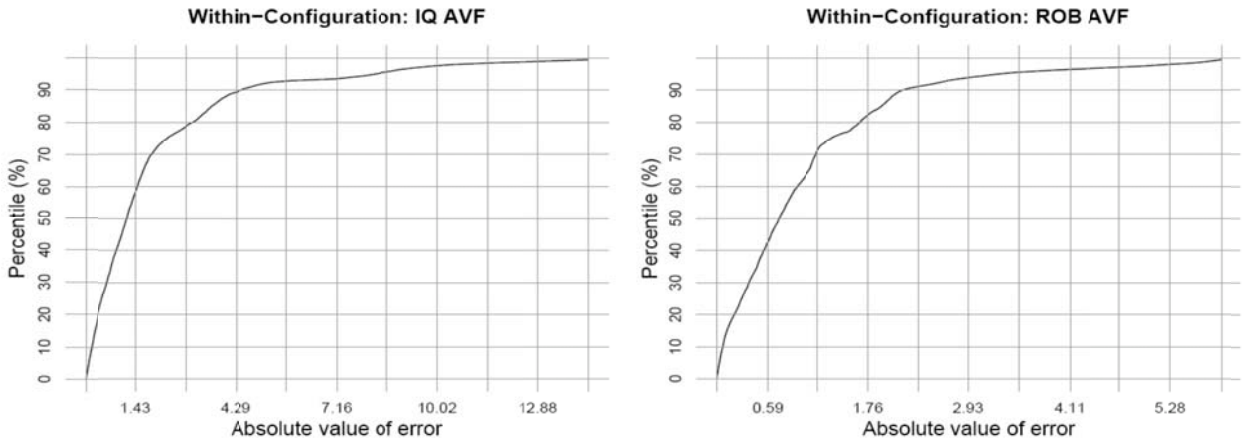

Figure 3-4. Empirical CDF on absolute errors in the with-configuration study

3.3.2 Prediction across Different Processor Configurations

The previous subsection demonstrates a correlation between the AVFs and a small set of performance metrics across workloads and phases but within a specified processor configuration. In this subsection, we further extend our methodology to address the cross-configuration situation. Applying the well-trained cross-configuration predictive model on different existing configurations enables calculating the AVF without implementing the various hardware required to calculate the AVF in each configuration. An analysis window is implemented to store 40K instructions after their commit stage. In order to detect the instruction type which will be used in calculating the AVF, we also need to maintain the instruction dependencies in the window by implementing a large amount of additional structures. Besides, a lot of state bits and control logics are expected in the AVF calculation. Therefore, if we have a cross-configuration predictive model, we can accurately quantify the AVF behavior on various hardware platforms without actually implementing the AVF calculation in hardware, thereby saving large hardware overheads.

Specifically, we tune the four parameters listed in Table 3-4 to generate 15 different configurations because these parameters are dominant in producing the vulnerabilities of IQ and ROB. Note that *cfg1* is the baseline setting described in Table 3-1. We still employ the BRT methodology to perform the prediction in this case. However, in order to characterize the change in configuration, we also include the tuned parameters in the performance metrics set as additional variables. Two randomly selected workloads, each also containing two phases, are simulated under each configuration. The training set consists of the phases under *cfg1* to *cfg12* (48 phase files in total) while the test set is composed of the other 3 configurations (12 phase files).

Table 3-4. Configurations used in Section 3.3.2. The training set contains the 48 phase files of *cfg1* to *cfg12* (white), and the test set includes the 12 phase files of *cfg13* to *cfg15* (gray).

	Fetch/slot/map /issue widths	Commit width	Issue queue size	Reorder buffer size	Simulated workloads (2 phases for each)
cfg1	4	11	20	80	*mcf, vpr*
cfg2	4	11	40	40	*eon.cook, gap*
cfg3	4	11	30	60	*crafty, perlbmk.makerand*
cfg4	4	11	40	80	*eon.cook, eon.rushmeier*
cfg5	2	7	20	80	*eon.kajiya, gap*
cfg6	2	7	40	40	*gcc.166, gcc.200*
cfg7	2	7	20	40	*gcc.expr, gcc.integrate*
cfg8	2	7	40	80	*gcc.scilab, mcf*
cfg9	1	3	20	80	*parser, perlbmk.makerand*
cfg10	1	3	40	40	*twolf, vortex.lendian1*
cfg11	1	3	20	40	*vortex.lendian2, vortex.lendian3*
cfg12	1	3	40	80	*vpr.route, bzip2.source*
cfg13	4	11	20	40	*bzip2.source, crafty*
cfg14	2	7	30	60	*eon.kajiya, twolf*
cfg15	1	3	30	60	*gcc.expr, vortex.lendian1*

Similar to the within-configuration study, we first apply BRT using all 217+4 input variables, and select the most important 10 features. After that, we refit the BRT model using the 10 metrics. The relative variable influences for this case are quantified in Figure 3-5. Interestingly, the structure's occupant rate becomes the most important variable to its AVF. We also observe that two configuration parameters (*issue_q_size, rob_size*) appear in the lists, indicating that changing configuration does have some impact in producing the AVF. As can be seen, the varia-

ble importance distribution (percentage and ranking) shown in Figure 3-5 is quite different from those depicted in Figure 3-1. This happens due to the multicollinearity problem in multiple regression models [34]. When many correlated input variables exist, the estimate of variable coefficients and their importance can be unstable since the effect from one variable may be disguised by its correlated variable(s). For example, *rob_average_count* and *rob_occupant_rate* are two highly corrected variables, but show completely different influences to the response in these two figures.

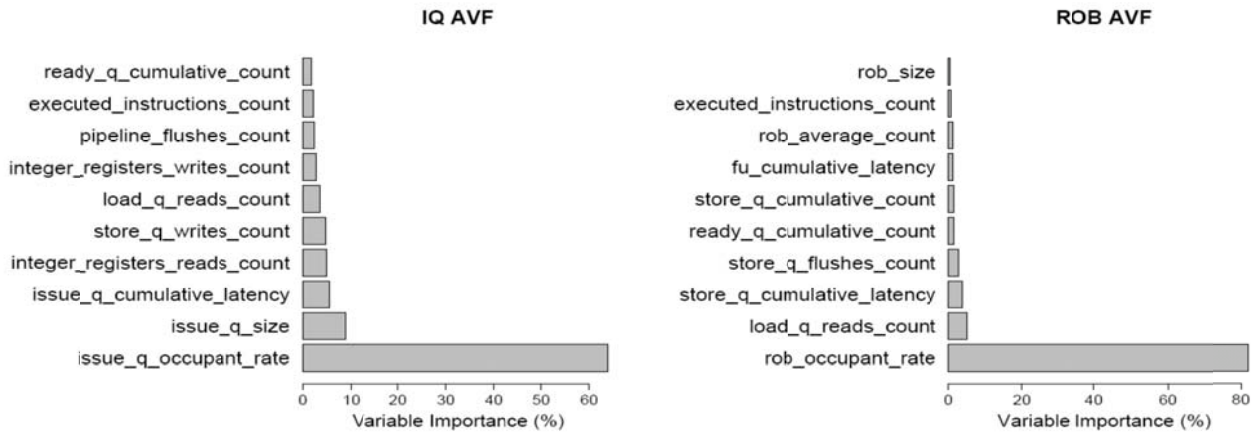

Figure 3-5. Input variable importance (across different configurations)

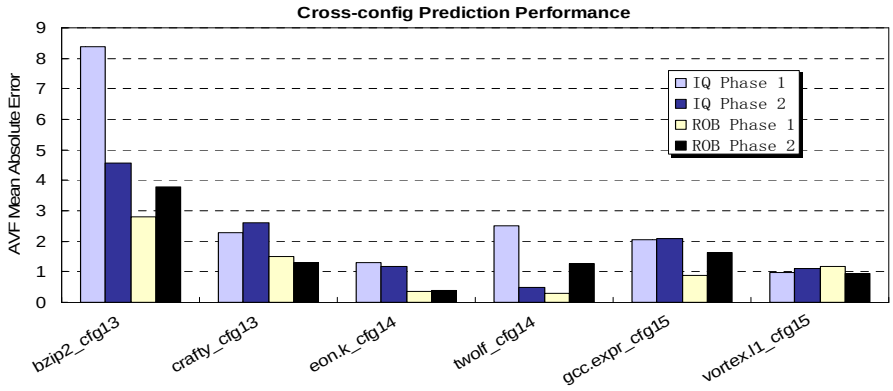

Figure 3-6. Prediction results on different configurations

As for the prediction performance illustrated in Figure 3-6, only one workload (*bzip2* under *cfg13*) is predicted with mean absolute errors of AVFs above 3, and the other 5 workloads in test show very high prediction accuracies in both IQ and ROB AVFs. Specifically, the overall MAEs of all the 6 workloads in the test set are 2.91 for IQ AVF (Phase 1), 2.0 for IQ AVF (Phase 2), 1.17 for ROB AVF (Phase 1), and 1.56 for ROB AVF (Phase 2). Note that the 6 workloads are simulated under 3 different configurations which also differ from the configurations in the training set. Hence, the accurate prediction results validate that our model is capable of predicting vulnerability behavior across configurations. In addition, Figure 3-7 shows that over 90% of the intervals are predicted below absolute errors of 4.6 and 2.1 for IQ and ROB AVFs, respectively.

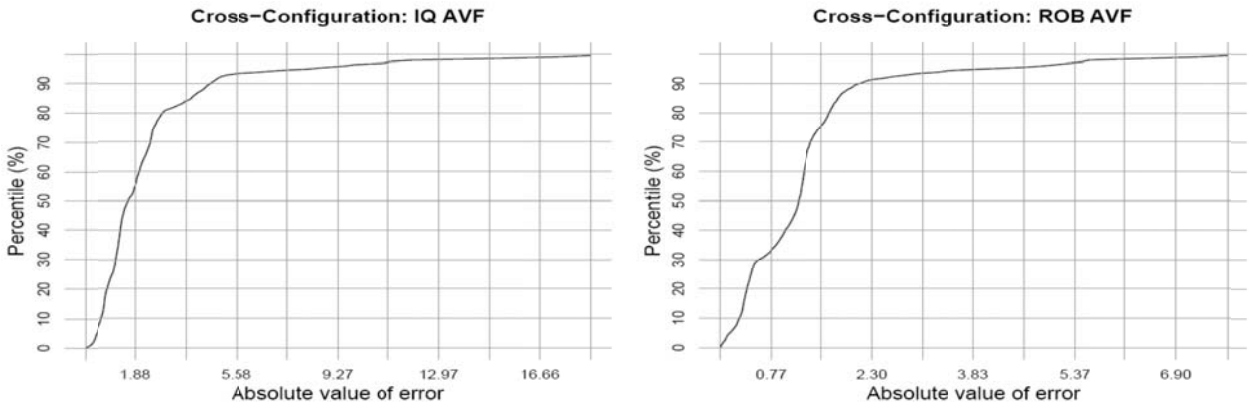

Figure 3-7. Empirical CDF on absolute errors in the cross-configuration study

3.3.3 AVF Behavior Analysis and Model Interpretation

Figure 3-8 provides another approach to compare the predicted and measured IQ AVF curves for two randomly selected workloads *gcc.integrate* and *crafty*. Although the measured AVF behavior shows extremely strong variation over time, our prediction is able to faithfully capture this behavior. Additionally, one can refer to Figure 3-9 and 3-10 for the partial dependence plots of the AVFs on the most important variables. As described in Section 2.2, Partial De-

pendence Function summarizes the effect of a subset of variables on the response (i.e. the AVF) after accounting for the average effect of other variables in the model. Therefore, partial dependence of the AVF provides computer architects with visible interactions between important performance metrics, implying the vulnerability trends and bottlenecks.

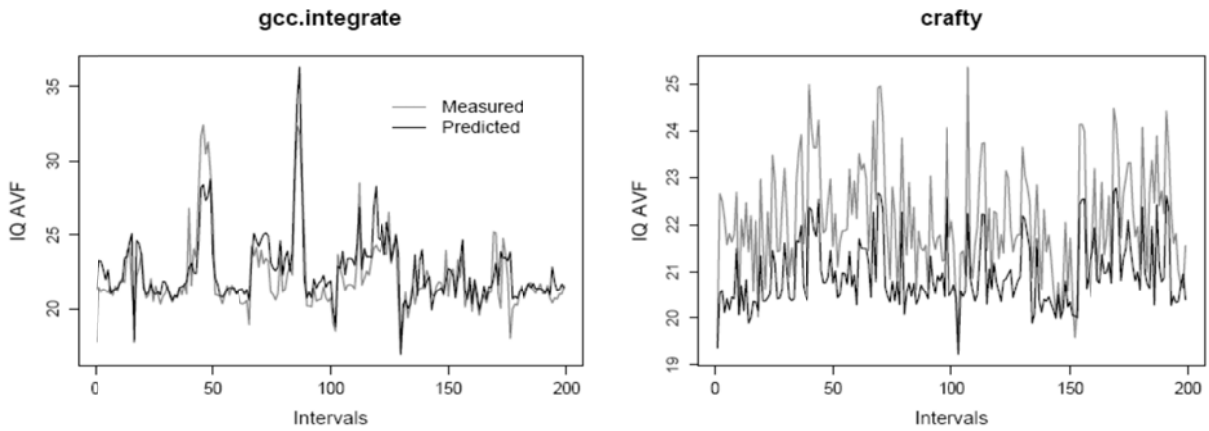

Figure 3-8. Measured and predicted IQ AVF curves for gcc.integrate and craft in the within-configuration study

Specifically, Figure 3-9 illustrates how the two most important variables contribute to the IQ AVF in the within-configuration study. The data is plotted into a contour map, which shows two hills where variation of the parameters results in significant changes of the AVF, and one plateau where the AVF is insensitive to the variables' changes. As can be seen, when the *issue_q_cumulative_latency* (the Y axis) is less than 5.8e+06, increasing this latency boosts the AVF from 18 to 24. In addition, when the *ready_q_cumulative_count* (the X axis) is less than 350K, decreasing the cumulative Ready Queue count leads to a further AVF increase from 26 to 36. A larger cumulative IQ latency means that the ACE instructions were kept for a longer time in the IQ. A lower Ready Queue count indicates a worse congestion in the IQ where the issued instructions wait for their operands to be ready. Both of these two cases contribute to a higher vulnerability of the IQ. The gray area in this figure represents a plateau where variations of the two metrics rarely affect the AVF. In this case, other processor variables should be considered.

In [25], the authors also used a nonparametric model and contour maps to analyze the roughness and bottlenecks of processor design topologies.

The proposed model can also quantify the AVF's partial dependence to one very important variable. The contribution of the ROB average count to the ROB AVF is shown in Figure 3-10. We can observe that the increase of the ROB average count results in the increase of the ROB AVF. This can be easily explained as the proportion of the valid ROB entries approximates the vulnerability of the ROB. The vulnerability saturates at around 23 when the ROB average count exceeds 48, in which case the ROB average count is no longer a driving factor to the AVF and other variables should be considered.

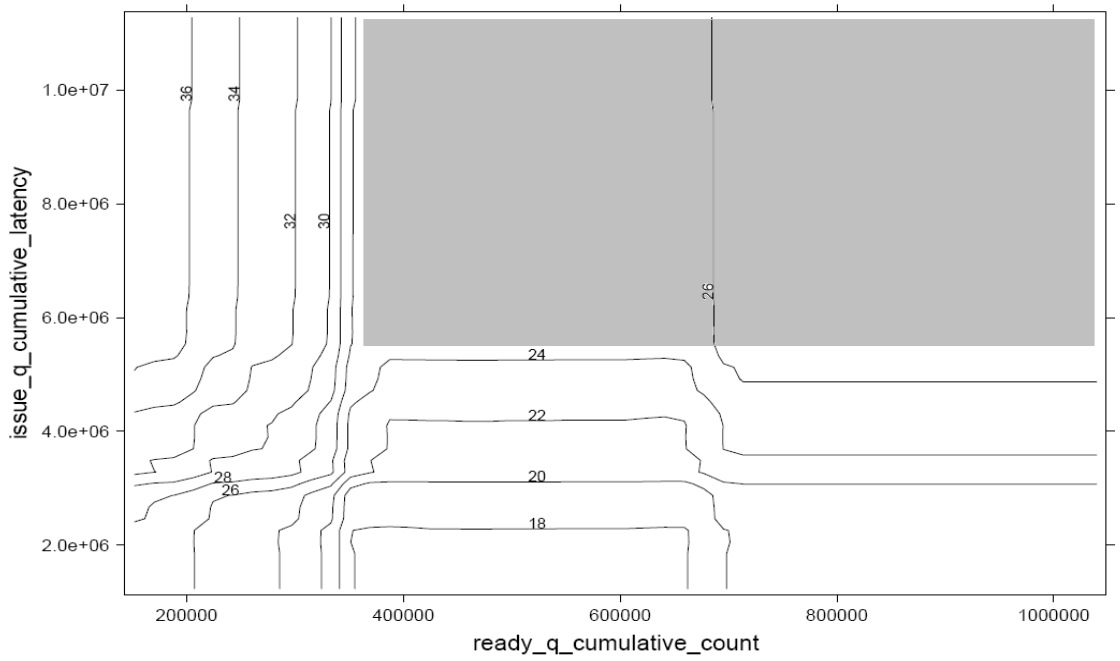

Figure 3-9. Partial dependence of the IQ AVF on the two most important variables in the within-configuration study

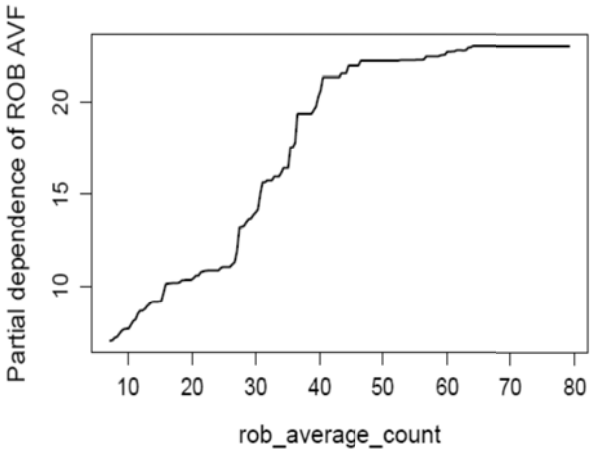

Figure 3-10. Partial dependence of the ROB AVF on the most important variable in the within-configuration study

3.3.4 A Comparison between BRT and Linear Regression

In this subsection, we make a quantitative comparison between our suggested BRT method and classical linear regression approach. For linear regression, we followed Walcott et al.'s approach [47]. Their proposed practical linear procedure started from including the single variable with the largest correlation into the model. Then by considering all the remaining variables (omitting the selected one) in turn, they selected the one that achieves the best bivariate linear regression approximation involving the previously selected variables. The procedure continues until all variables are included in the model.

Figure 3-11 illustrates the comparison. Here we only consider the IQ AVF prediction for the within-configuration case. The left panel shows the R-squares on the training and test sets with different numbers of variables included in the model in the linear regression approach and our BRT method. We see that although the R-squares increase monotonically on the training set for both methods, the test R-squares do not. In linear regression, the test R-square goes below zero when five to eight variables are included in the model. Therefore, the test R-square in BRT is

more stable than linear regression. This is also exemplified in the right panel of this figure, which shows the coefficient solution paths along the model size in linear regression. Notice that for "X3", its estimated coefficient begins with a negative value at model size 3, shrinks towards zero at model size 4 & 5, changes to positive at model size 6 & 7, and goes below zero again for model size 8 to 10.

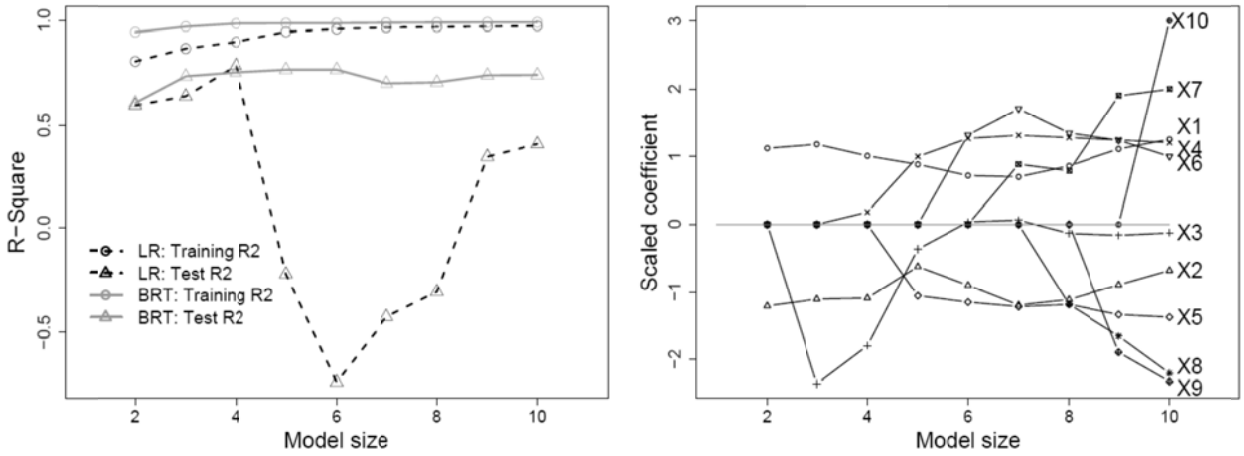

Figure 3-11. R-squares on training and test sets over model size in linear regression and BRT methods (Left), and the scaled coefficients for the first ten selected variables in linear regression (Right).

The instability issue in model fitting and estimation has been well studied in [6] where it pointed out that neural nets, classification and regression trees, and subset selection in linear regression were unstable. Instability refers to the situation that a small change in input data set (i.e. including/excluding one workload or phase) or model setting (such as model size) can result in a significant change in the fitted model (i.e. estimate of model coefficient and fitted value). As demonstrated previously, linear regression suffers from such instability issues. One common approach to alleviate instability problem is model averaging. As we mentioned earlier, regression trees are the building blocks for BRT. Although they are highly greedy and instable optimization methods, BRT averages a large number of regression trees with a tiny equal weight ν (ν=0.01)

on each tree. This stabilizes the estimate from BRT and achieves better prediction performance than a single regression tree. Figure 3-12 shows the R-square curves for training and test sets with ν equals to 0.01 and 1 in the BRT model. The benefit of using small value of ν is evident. With a smaller value of ν, the test R-square curve reaches a higher value and stays there for many iterations. In other words, our BRT model has successfully stabilized the prediction.

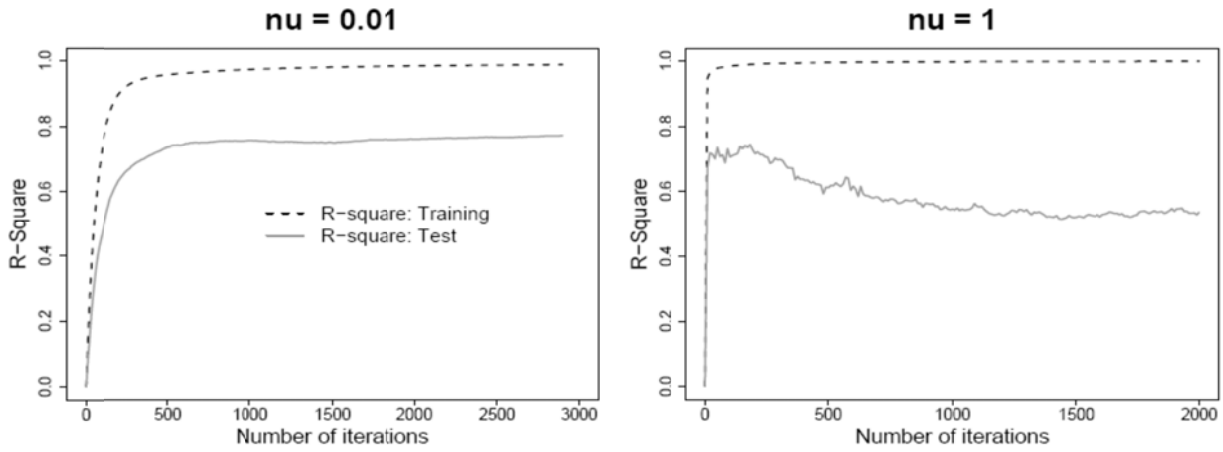

Figure 3-12. R-square curves for the training and test sets with different learning rate ν in BRT

In addition, BRT is a more flexible modeling scheme than linear regression. For example, it considers nonlinear relationship between the response and input variables, addressing complex interactions among input variables. If we consider the nonlinearity and interactions in the linear regression model, we have to specify each term carefully and do the model checking before getting the final model. Our method also has model interpretations which help us visually analyze the vulnerability trends and bottlenecks. This cannot be done by a linear regression model.

3.4 Fast AVF Estimation

In practice, a simpler AVF prediction mechanism is easier to be adopted. In order to reduce the model complexity, we further propose to use a PRIM-based technique to summarize some simple and interpretable "IF-ELSE" rules that can be applied to some important performance variables during runtime to quickly identify the intervals with high AVF values. We demonstrate the effectiveness of this method by illustrating the ROB AVF prediction results within the baseline configuration.

The results of fast ROB AVF estimation in the within-configuration study are shown in Figure 3-13. We intend to find the top ~10% of the intervals in terms of the vulnerability level. Note that we denote a high vulnerable interval as a black "o" while an interval with a low vulnerability as a gray "+" in this figure. The training and test sets are the same as those in Section 3.3.1; that said, the training set shown in the left part of Figure 3-13 contains 3,000 intervals (white columns in Table 3-2) while the test set contains 4,600 intervals from the benchmarks and phases listed in gray columns of Table 3-2. The rules extracted from the training data can be described in Figure 3-14.

One can refer to Table 3-3 for the explanation of variable names. The only one here that was not listed in Table 3-3 indicates the cumulative latency that the committed instructions spent in passing the whole pipeline. From the testing results shown in the right part of Figure 3-13, we can see that applying these simple rules to the test set makes an accurate AVF estimation, i.e. the AVF of the current interval is high or not. The derived rules can be explained from an architectural perspective: the valid ROB entries and the cumulative latency to go through it perform the estimation in the first place; longer cumulative slip latency reflects a lower instruction processing

speed of the whole pipeline; infrequent writes to the Load Queue also make the vulnerable instructions stay long in the pipeline. Hence, all the identified rules show strong significance in estimating the architectural vulnerability.

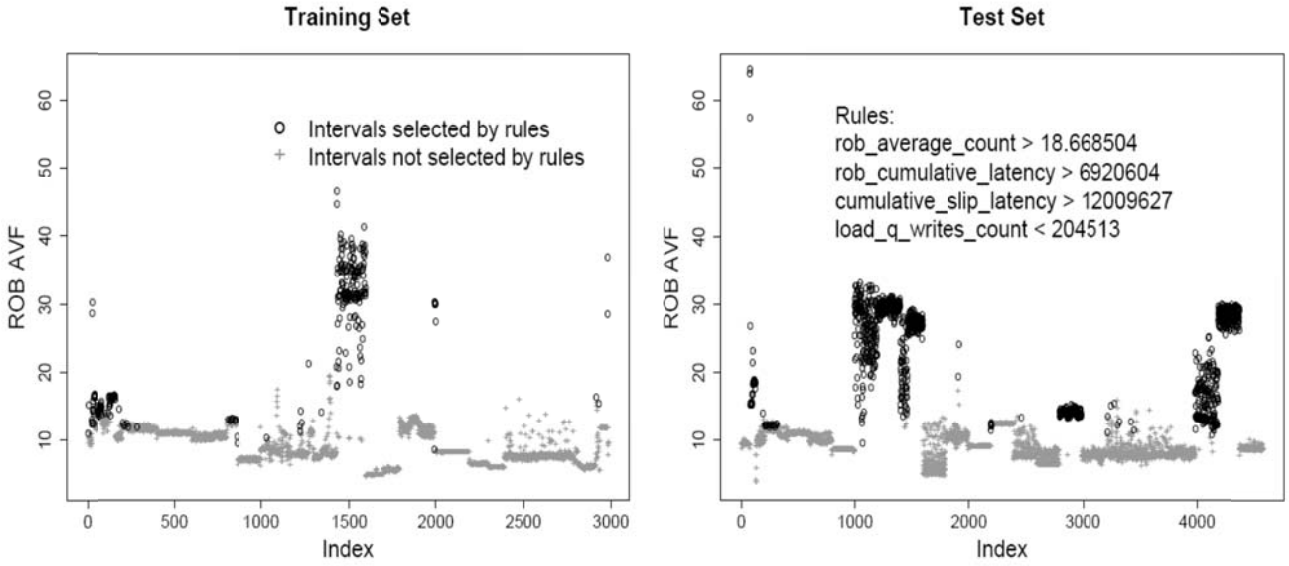

Figure 3-13. Fast estimation of the ROB AVF in the within-configuration study

```
IF      ((rob_average_count > 18.668504)
AND   (rob_cumulative_latency > 6920604)
AND   (cumulative_slip_latency > 12009627)
AND   (load_q_writes_count < 204513))
THEN {
        The interval is declared to have a high ROB-AVF value
}
```

Figure 3-14. Derived rules for the ROB AVF estimation

3.5 Case Study: PRIM-Based ROB Redundancy

This section describes am application of the PRIM-based scheme introduced in Section 3.4 to efficiently bind the ROB AVF value under a system-affordable low level. While most inter-

vals shown in Figure 3-13 demonstrate a very good invulnerability to transient faults in the ROB, some of them (around 10% in the entire workload space) do have a relatively high ROB AVF that may result in a failure against system reliability requirements. However, as can be seen in the figure, our PRIM-based model is able to effectively identify such vulnerable intervals, in which we can essentially partition the ROB into two identical halves that execute the same instruction flows, generating necessary redundancy to detect possible ROB transient faults. Similar studies can be found in [41][47], but ours does not bound the AVF below some pre-specified threshold; instead, we systematically analyze the vulnerability distribution in the entire workload space, and generate some simple rules that select the intervals of interest, i.e. the vulnerable intervals showing a high AVF. This section mainly focuses on reducing the ROB AVF in the within-configuration study, but the approach can be easily applied to the cross-configuration situation or other processor structures.

3.5.1 The Approach

The basic idea of our approach is that whenever the program execution finishes an interval whose measured performance metrics conform to the trained PRIM rules (i.e. the interval is declared to be vulnerable in the ROB), we partition the ROB into two identical halves, both running the same instruction flows from the next interval. This is defined to be in a *redundant mode*, during which the ROB AVF is effectively reduced to zero due to the generated redundancy. After a fixed number of intervals, the ROB will switch back to a *normal mode* which recombines the two parts into the original buffer, and start checking the metrics again. Figure 3-15 details the algorithm; *redundancy_flag* records how many intervals remain in the redundant mode to reach the next normal mode interval.

```
At the beginning of each interval:
        IF      ((redundancy_flag = 0) AND
                (ROB in the redundant mode))
        THEN {
                        ROB switches to the normal mode
                }

At the end of each interval:
        IF      ((redundancy_flag = 0) AND
                (current measured metrics conform to the PRIM rules))
        THEN {
                        ROB switches to the redundant mode
                        set redundancy_flag to n
                }
        IF      (redundancy_flag > 0)
        THEN {
                        decrement redundancy_flag by 1
                }
```

Figure 3-15. PRIM-Based ROB redundancy algorithm

To switch the ROB from the normal mode to the redundant mode, we need to properly process the valid instructions that still remain in the ROB such that the execution correctness is preserved. This can be done via *Fetch Throttling*, which is similar to *Dispatch Throttling* [41]. When a vulnerable interval is detected, we temporarily stop fetching instructions until the whole pipeline is drained and the ROB finishes the mode switching. After that the pipeline resumes instruction fetching, and every instruction will be placed in both ROB partitions to enable the redundancy. This endows the ROB with the ability to detect possible transient faults.

When the ROB is in the redundant mode, the effective ROB size reduces to one half of the original size, and the PRIM rules that were derived in the normal mode become inappropriate. Therefore, we do not follow those selective rules under the redundant mode, but simply switch the ROB back to the normal mode after n intervals. In this study, we fix n at 10. Note that this scheme may potentially disable the ROB redundancy even when its AVF of current interval turns out to be high (but will re-enable the redundancy immediately after current interval if this hap-

pens), resulting in some "outliers" escaping from our bounding effect. This is considered to be negligible since such outliers rarely exist in our 200-interval workload executions.

3.5.2 Performance Degradation Analysis

Our PRIM-based ROB redundancy may suffer from performance degradation due to two reasons. One is that the effective ROB size is reduced to half in the intervals under the redundant mode, and the other is that we need to throttle fetching instructions to drain the pipeline when the normal-to-redundant mode switching happens. We analyze the performance degradation in this subsection to demonstrate that our redundancy scheme sacrifices only a negligible amount of performance to provide significant reliability improvement.

The average IPC degradation after applying our ROB redundancy scheme in within-configuration study is 1.9% for the 23 workloads in the test set. There are only 3 of them showing a relatively significant IPC decrease, while the rest 20 retain their performance very well. For analysis purpose, we only present two typical examples here: one is from the former group that suffers from performance degradation, and the other is from the latter ones that retain the IPCs.

gcc.scilab (phase 2), whose IPC decreases by 9.3% when enabling the redundancy scheme, is one of the 3 workloads that suffer from a noticeable IPC degradation. Figure 3-16 shows its IPC variation before and after applying PRIM-based ROB redundancy. An apparent two-step curve can be seen from this figure: the IPC retains well in the first step but nearly decreases by a constant amount in the second step. One can refer to Figure 3-18 for an easy explanation: the ROB occupant rate (for *gcc.scilab*) hangs around 50% in the first step but reaches 80% in the second step, where the system would be short of the ROB entries if the ROB size is cut to half. In contrast, *bzip2* (phase 2) only presents a 0.2% decreasing in the IPC, whose variation is de-

picted in Figure 3-17. Note that the two curves are highly overlapped with only a few points in the middle showing slight difference. Figure 3-18 also gives the ROB occupant rate for *bzip2*, and it's not surprised to see a well retained IPC in this workload since its ROB occupancy persistently stays below 50%.

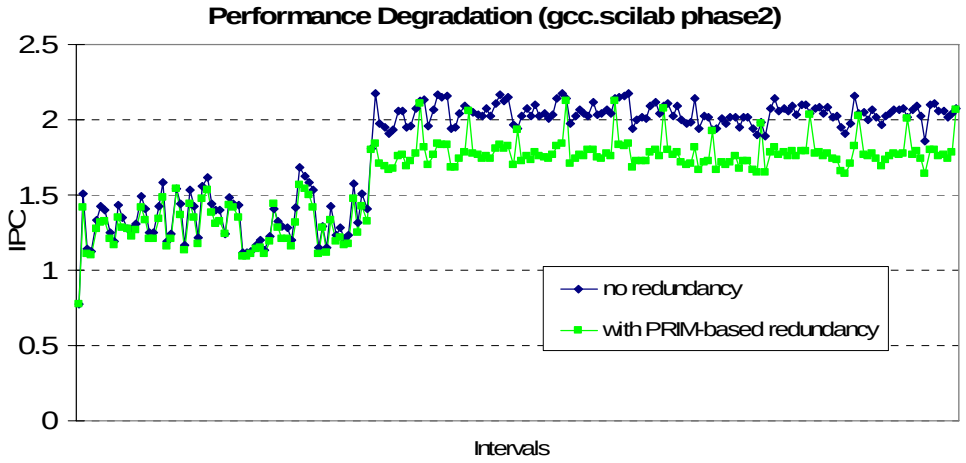

Figure 3-16. IPC variation for *gcc.scilab* in phase 2 of both schemes: without ROB redundancy and with PRIM-based ROB redundancy.

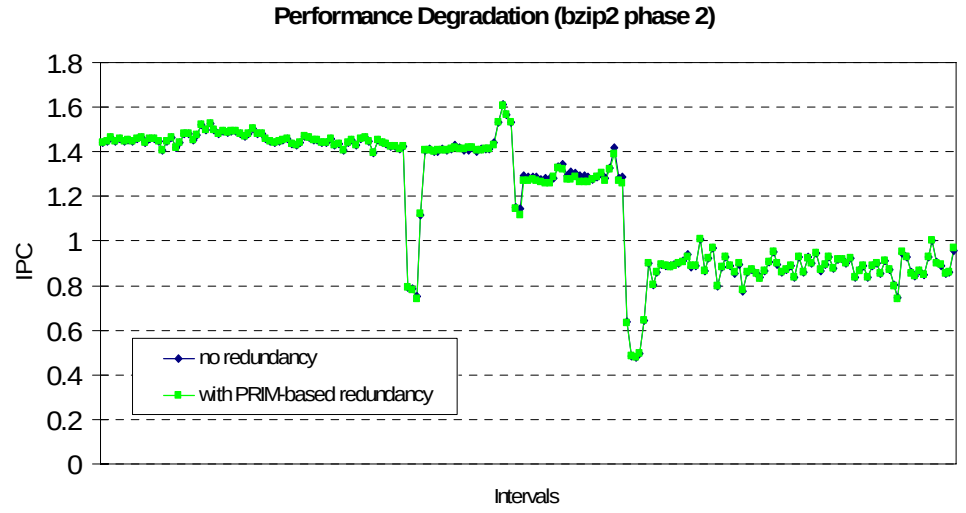

Figure 3-17. IPC variation for *bzip* in phase 2 of both schemes: without ROB redundancy and with PRIM-based ROB redundancy.

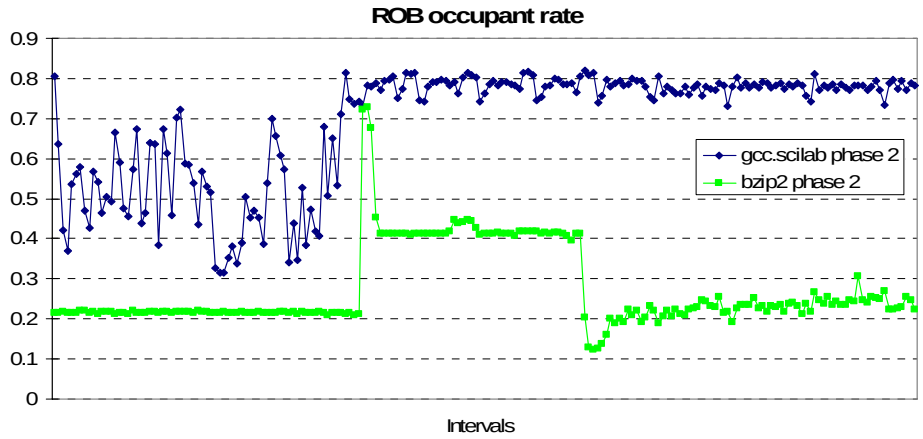

Figure 3-18. ROB occupant rate for *gcc.scilab* in phase 2 and *bzip2* in phase 2.

3.6 Related Work

Mukherjee et al. [32] compared the advantages and disadvantages of three different RMT techniques: (1) Lockstepping, a cycle-by-cycle synchronization that has long been used on commercial fault-tolerant systems; (2) Simultaneous and Redundantly Threading (SRT) [35], which utilizes the dynamic resource sharing from SMT processors to reduce performance degradation due to redundancy; and (3) Chip-Level Redundant Threading (CRT), which extends SRT to a CMP environment, explores significant performance benefit on multithreaded workloads. Vijaykumar et al. [46] and Gomaa et al. [20] proposed the recovery schemes for SRT and CRT, respectively. Prior to these schemes, Rotenberg [36] and Austin [1] at first proposed different transient fault detection architectures.

There are two main approaches to calculate the AVF: ACE analysis and Statistical Fault Injection (SFI). The former provides a (tight, if the underlying system is appropriately modeled [4]) lower bound on the reliability level of various processor structures, and has been adopted in many research works on performance models. Fu et al. [19] quantitatively characterized vulnerability phase behavior of four microarchitecture structures based on a system framework [18].

Zhang et al. [52] performed a similar analysis on SMT architectures. Soundararajan et al. [41] described a simple infrastructure to estimate an upper bound of the ROB AVF, and also proposed two mechanisms (Dispatch Throttling and Selective Redundancy) to restrict the vulnerability to any limit.

Alternatively, SFI randomly (or statistically) injects into program execution a set of faults, each being independently analyzed and determined to see a visible error of the outcome. The AVF is the ratio of the number of trials that eventually raise an error to the total number of trials performed. Wang et al. [49] implemented a latch-accurate Verilog model to simulate an Alpha processor while Li et al. [29] incorporated a similar probabilistic model of error generation and propagation into an architecture-level tool. Wang et al. [48] compared ACE analysis to their fault-injection IVM, and claimed that ACE analysis was highly conservative by identifying two sources of its conservatism (lack of system detail and single-pass simulation). However, a recent publication [4] refuted their claim by stating that a small amount of additional details can result in a much tighter AVF bound and quantifying the small effect of Y-bits on system simulation.

Besides [47], some other works also addressed the problem of runtime AVF prediction. Cho et al. [8] examined workload dynamics in a design space of microarchitecture configurations. For each workload, they trained a set of neural networks with series of wavelet coefficients decomposed from AVF behaviors under different configurations, predicted the wavelet coefficients of any other configuration, and reconstructed the AVF curve (of the target configuration) from the predicted coefficients. Their work is completely different from ours in this chapter because they required a separate (or different) set of neural networks for each workload while our model has been demonstrated to be validated across workloads, phases and configurations. Very recently, Li et al. [28] developed an algorithm to estimate processor structures' vulnerability online using

a modified error injection and propagation scheme from their previous work [29]. Their method does not need any offline simulation (except some experimental experience to determine key parameters), but requires hardware modification of the processor to support error propagation and detection rules.

3.7 Conclusions

In this chapter, we proposed to use Boosted Regression Trees, a nonparametric tree-based predictive modeling scheme, to identify the correlation (across different workloads, execution phases, and processor configurations) between a key processor structure's AVF and various performance metrics. Experimental results showed that our model can accurately predict the AVF in the above situations. In addition, the proposed model provides useful interpretation tools for computer architects to quantify important variables and the AVF's dependence on them. A quantitative comparison between the BRT model and linear regression demonstrates that our scheme is more stable and has many advantages. Finally, to reduce the prediction complexity, we also utilize another technique named Patient Rule Induction Method to extract some simple selective rules to monitor a few important metrics, which can be used to quickly identify the execution intervals with a relatively high AVF. The case study performed in the chapter also justifies the applicability of our fast AVF estimation scheme.

CHAPTER 4. AVF PREDICTION ON MULTI-THREADED PROCESSORS

4.1 Summary

Architectural Vulnerability Factor (AVF) provides useful guidelines in designing reliable processors, but its measurement is extremely expensive in terms of hardware and computation. This is precisely the reason that we proposed AVF prediction from performance metrics in the previous chapter. However, the prediction was conducted only on a single-threaded superscalar processor; the AVF measurement on a multi-threaded processor, e.g. Simultaneous Multithreading (SMT) or Chip Multiprocessor (CMP), is much more complicated and involved. In this chapter, we characterize and predict a program[2]'s soft error vulnerability under resource contention and sharing with other programs running on SMT/CMP architectures.

Difficulties. First, inter-thread resource contention and sharing significantly and non-uniformly affect the AVF of structures[3] of different hardware threads. Figure 4-1 shows different processor structures' AVF variation when *gcc* runs alone and against different benchmarks on a two-way SMT and a dual-core CMP (which are based on the same processor configuration for comparison purposes). ROB refers to reorder buffer, and IQ refers to instruction queue. We can see that the impact of resource sharing is relatively small in the CMP, whose inter-thread contention is mainly located in the shared cache but has little impact on the processor structures' AVFs.

[2] In this chapter, a program refers to a single-threaded program. We use program and thread interchangeably. A multi-programmed workload refers to a program combination, and a multi-threaded workload refers to a program with multiple threads that have data sharing.

[3] In this chapter, AVF always refers to the AVF of hardware structure(s), which can be private to some thread or shared among threads. Sometimes, we use a thread (program)'s AVF to refer to the AVF of processor structures that are private to the thread.

On the other hand, the contention resulted from pipeline resource sharing in an SMT processor significantly and non-uniformly affects the AVFs when *gcc* is co-scheduled with different benchmarks. In addition, despite the strong variation, ROB AVF of SMT is consistently lower than that when *gcc* runs alone, but IQ AVF may be higher (e.g. *gcc+apsi*) or lower (e.g. *gcc+crafty*). This interesting observation introduces new issues into AVF behaviors (compared to contention-free superscalar's AVF), indicating higher difficulty in accurately predicting a multi-threaded processor's AVF.

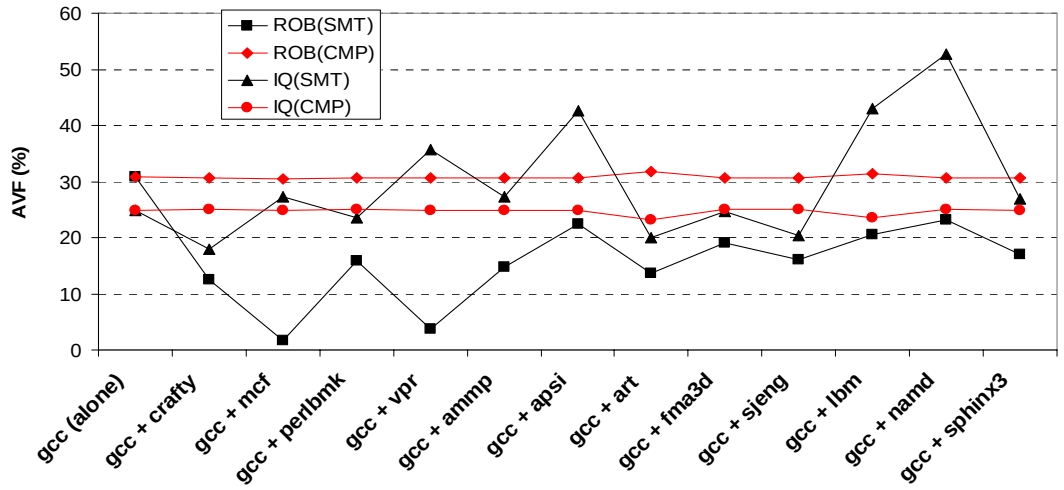

Figure 4-1. The AVF variations of gcc (SPEC 2000) when it is co-scheduled with different benchmarks on SMT/CMP

Second, the problem complexity and scope are significantly enlarged in the context of multi-threading. AVF reflects soft error masking at both program level [44] and machine level [45], so AVF prediction should take into account both applications and processor configurations. Walcott et al. [47] performed the prediction across SPEC CPU 2000 benchmarks on a fixed machine configuration; our first work [11] in this dissertation extended their prediction to be across a very small set of configurations by changing only 4 parameters. Nevertheless, both of their works were restricted to single-threaded processors with certain simplifications. In contrast, this work

intends to correlate these two important (but complex) problems: the processor configuration being from a statistically large design space, and the prediction effectiveness across different multi-programmed and multi-threaded workloads. Furthermore, the predictive mechanism proposed in this chapter provides an efficient (and thereby practical) approach for multi-threading AVF prediction. Traditional application-specific design space studies build a separate model for each program. This will result in intractable training cost in multi-threading environments since multi-programmed workloads produce combinatorial growth in the number of possible program combinations [26]. Our AVF predictor's cross-workload capability (i.e. only a single model for all workloads) is extraordinarily useful for efficient predictions on multi-threaded processors.

Our proposal: two-level predictive modeling. In this work, we propose a two-level predictive mechanism capable of accurately predicting key processor structures' soft error vulnerability on multi-threaded processors with resource contention and sharing. At the first level, a cross-program model is trained to predict the contention-free AVF on a single-threaded processor. The inputs to the first level model include a few important performance measurements (e.g. structure occupancies, cache miss rates) from the contention-free execution and the corresponding configuration parameters. The output of the first level model, along with key processor structures' occupancies measured when the program runs against other program(s) on a multi-threaded processor, are inputted to the second level model, which finally predicts the program's AVF under resource sharing with others.

Essentially, the first level model uses key parameters and simple performance measurements to characterize underlying hardware configuration and the software program, respectively; while the second level model captures the inter-thread resource contention and sharing. Note that both of the two levels are "universal" for different programs, taking some performance metrics

as part of the inputs. The difference is that the performance metrics at the two levels are measured from single-threaded execution (without contention) and multi-threaded execution (with contention), respectively. By employing the proposed two-level prediction, we can obtain an accurate estimation of a program's AVF when it is co-scheduled with different programs on an arbitrarily configured SMT/CMP whose configuration is from a design space.

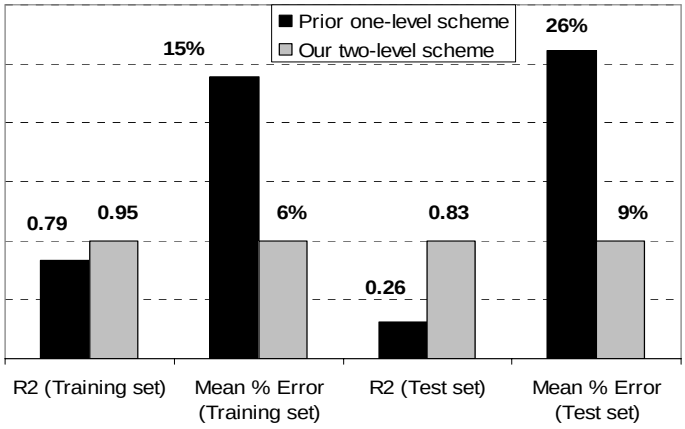

Figure 4-2. Comparison between the proposed two-level scheme and prior one-level scheme

Why are two levels necessary? Simply applying prior works' [47][11] one-level approach fails to make accurate predictions on multi-threaded processors. For demonstration, we trained a one-level model to predict SMT processors' register file AVF. We used the same prediction technique as in [11] since it demonstrated better performance than linear regression used in [47]; the inputs to this model are the various performance measurements when different threads are run simultaneously. For a fair comparison, the training and test sets are the same for these two approaches. Figure 4-2 compares the prediction accuracy between the two models in terms of R-square (higher is better with 1 as the maximum) and mean percentage error (lower is better with 0% as the minimum). We can see that the one-level scheme shows more than two times higher error rates than the two-level scheme. Especially for the test set, the mean percentage error

reaches a very high value (26%) while the R-square is unacceptably low (0.26). Essentially, decoupling the prediction into two levels reduces the model complexity at each level and also improves prediction accuracy.

Contributions. In summary, the main contributions of this chapter are as follows:

- **Universal prediction of the AVF on single-threaded processors**. The first level model accurately predicts the AVF without contention on any given processor configuration in the design space. This model is universal for different programs, and also validated for unseen programs not used in training.
- **Universal prediction of the AVF under contention across multi-programmed workloads**. The second level model takes the knowledge of the contention-free AVF from the first level model, and performs an accurate prediction of the AVF under resource contention for any program combination in analysis. This combined capability is extremely useful in the era of multi-threading.
- **AVF prediction of multi-threaded workloads with data sharing**. This chapter discusses the impact of data sharing among different threads on the AVF of multi-threaded workloads.
- **A case study of soft error resilient thread-to-core scheduling**. By utilizing the proposed AVF prediction, we also present a case study that identifies the optimal thread-to-core assignment minimizing the AVF of a Chip Multi-Threaded Processor.

4.2 Two-Level Prediction Methodology

The ultimate goal of this work is to predict the AVF of a program in contention with other program(s) running simultaneously on a multi-threaded processor (e.g. SMT, CMP) without

AVF measurement mechanisms. It can be predicted from the AVF when this program runs alone without contention, and a group of important performance metrics reflecting the occupancy rates[4] of key processor structures. The latter can be easily measured during program execution on an SMT/CMP, but the former cannot. Therefore, we further predict the contention-free AVF from the underlying single-threaded processor. We organize the predictions in a two-level predictive model in this subsection. The Boosted Regression Trees (BRT) algorithm described in the previous chapter serves as building blocks to our two-level model.

At the first level, a universal model is trained to predict the AVF in a contention-free single-threaded processor. We first sample a group of training configurations from a huge processor design space, and then simulate them for benchmarks in the training set. The AVF and a few performance measurements (e.g. IPC, cache miss rates, and structure occupancies) are measured during simulation. The model at this level is trained with these performance measurements and the corresponding configuration parameters as the inputs and the contention-free AVF as the response. Figure 4-3 illustrates the training procedure. For brevity, only the inputs from a certain benchmark $\mathbf{B_k}$ are shown; but this procedure is actually performed for all the n training benchmarks.

For the second level model involving inter-thread contention, Figure 4-4 takes a benchmark combination $\mathbf{(B_i, B_j)}$ as an example to illustrate its training procedure. The AVF of $\mathbf{B_i}$ from single-threaded execution and a few processor structure occupancies measured from multi-threaded execution in which $\mathbf{B_i}$ competes with $\mathbf{B_j}$ serve as the inputs to the model; while the AVF of $\mathbf{B_i}$ in contention with $\mathbf{B_j}$ is the response to be predicted. We provide training examples from different benchmark combinations, making the second level model universal to different benchmark com-

[4] The occupancy rate of a processor structure is the proportion of its entries that are in use.

binations. The performance metrics at this level (i.e. structure occupancies) reflect the inter-thread contention and sharing; in contrast, those at the first level (in Figure 4-3) are measured from a contention-free single-threaded processor. All of these performance metrics are easy to be measured in either simulators or real processors via performance counters.

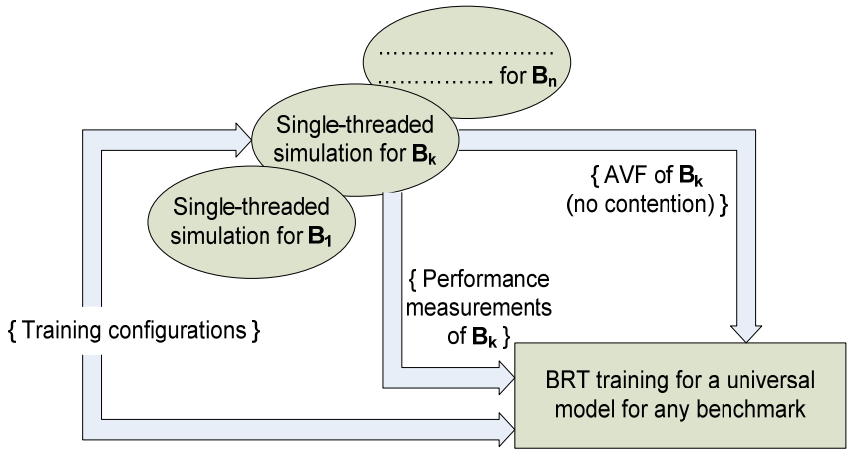

Figure 4-3. The first level universal model training using BRT. Only the inputs from a certain benchmark B_k are shown.

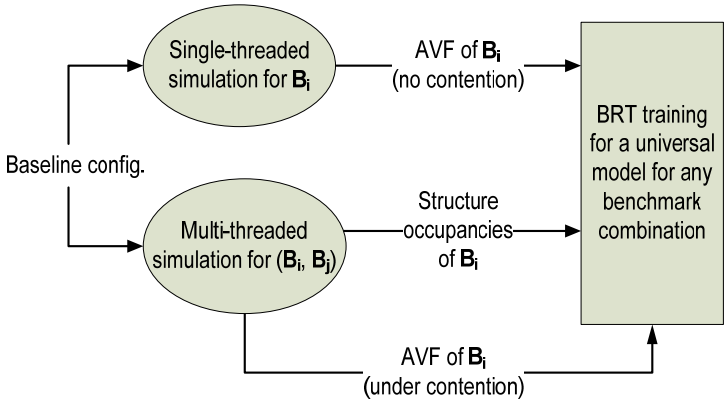

Figure 4-4. The second level universal model training using BRT

By combining the two levels, we will be able to predict the AVF on an SMT/CMP running any benchmark combination. As shown in Figure 4-5, in order to predict the AVF of a benchmark $\mathbf{B_k}$ (in particular, $\mathbf{B_k}$ can be any unseen benchmark not in the training set) running against

another benchmark, say $\mathbf{B_t}$, on a given processor configuration, we need to follow the following steps:

(1) Running single-threaded simulation for $\mathbf{B_k}$ with the processor configuration in analysis, and collecting the performance measurements after simulation.

(2) Giving the above performance measurements and the configuration parameters as inputs to the first level model, and then predicting the AVF of $\mathbf{B_k}$ without contention.

(3) Running multi-threaded simulation for the benchmark combination ($\mathbf{B_k}$, $\mathbf{B_t}$), and measuring key structure occupancy rates under contention.

(4) Providing the predicted contention-free AVF and the measured key structure occupancies as inputs to the second level model, and finally predicting the AVF of $\mathbf{B_k}$ under resource contention with $\mathbf{B_t}$. Note that, to predict for an unseen benchmark $\mathbf{B_k}$, the above approach only needs a single pass of single-threaded simulation for $\mathbf{B_k}$ and a single pass of multi-threaded simulation for $\mathbf{B_k}$ and its resource competitor.

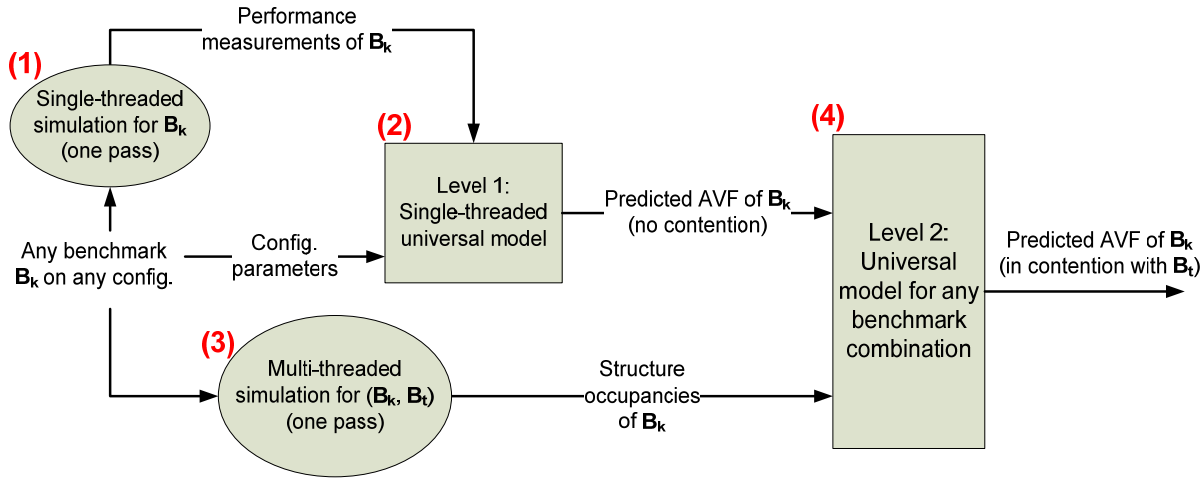

Figure 4-5. An overview of the two-level AVF prediction on a multi-threaded processor

4.3 Impact of Multi-Programmed Resource Contention on AVF

4.3.1 Experimental Setup

We implemented the AVF measurements [33] in M-Sim3.0 [37] to model the soft error vulnerability of key processor structures. In M-Sim's SMT model, each hardware thread has its own ROB and LSQ, but other structures (e.g. Issue Queue, Functional Units, Physical Register File, etc) are shared among threads. Basically, a fetcher (ICOUNT in our experiments) fetches instructions for each thread, and stores them in the corresponding ROB/LSQ after decode and rename; each thread dispatches instructions from its own ROB into a shared IQ in contention with other threads; the inter-thread contention exists in the remaining pipeline stages until commit. On the other hand, the CMP model creates a separate core (along with the private L1 I/D caches) for each thread, but the L2 unified cache is shared among cores.

In this work, we measure and study the AVF of the following five structures: Reorder Buffer (ROB), Load Store Queue (LSQ), Issue Queue (IQ), Functional Units (FU), and Physical Register File (RF). In the case of SMT, we will perform prediction and analysis separately for each of these structures, because the private structures (ROB and LSQ) and the shared structures (IQ, Functional Units, and RF) demonstrate different behaviors in the AVF under contention; in a CMP, since almost everything is private to each core, we will focus on the core AVF.

38 benchmarks from SPEC CPU 2000 and 2006 suites are evaluated. For the other SPEC 2000/2006 benchmarks not included here, we could not compile them into Alpha binaries runnable in M-Sim3.0. We first focus our work on two-way SMT and dual-core CMP, and then discuss model scalability to more than two threads. We first simulate these benchmarks on our single-threaded baseline configuration (whose parameters are shown in bold in Table 4-2), and cat-

egorize them in Table 4-1 according to the IPC and L2 cache miss rate. We use SimPoint toolkit [38] to derive a representative 100-million instruction phase for detailed simulation of each benchmark. The number of fast forwarded instructions is shown after the benchmark name in this table. In order to capture different CPU/memory behaviors, we randomly choose some benchmarks from each of the four categories for test (the 4th column of Table 4-1). Consequently, the 435 two-threaded combinations generated from the 30 training benchmarks (i.e. $C_{30}^2 = 435$) serve as the training examples to the predictors in the following sections. The predictors will be tested with the other 268 combinations, in which either both benchmarks are from the 8 test benchmarks or one is from the training set and the other is from the test set (i.e. $C_8^2 + 30 \times 8 = 268$).

Table 4-1. Benchmarks used in multi-programmed workloads. The number of fast-forwarded instructions (unit: 100M) is shown after the benchmark name.

IPC	L2 cache miss rate	Training benchmarks	Test benchmarks
> 1.0	< 30%	*gcc(135), sphinx3(12), apsi(382), eon(201), gzip(372), h264ref(272), hmmer(342), mesa(322), perlbmk(5), gap(324), gromacs(285), sjeng(498), 06bzip2(633), fma3d(150)*	*calculix(200), namd(45), vortex(570)*
< 1.0	> 30%	*mcf(142), art(23), swim(234), lucas(597), equake(408), vpr(292), ammp(283), astar(449), lbm(69)*	*milc(85), mgrid(236)*
> 1.0	> 30%	*wupwise(95), libquantum(200), applu(281)*	*facerec(208)*
< 1.0	< 30%	*galgel(415), crafty(113), bzip2(152), gobmk(217)*	*parser(270), twolf(176)*

We will discuss the second level model first in this section because we are more interested in the contention behavior. All the training data used in this section are directly measured from

the baseline configuration. In Section 4.4, this model will be combined with the first level model to demonstrate the effectiveness of the entire two-level predictive mechanism.

4.3.2 Impact on SMT Private Structures

ROB and LSQ are private to each thread in our SMT implementation. These two structures only contain instructions from a single thread, no matter whether it runs alone or competes with another thread. However, the inter-thread contention due to the shared pipeline resources strongly affects a thread's AVF for these two structures. We discuss the ROB AVF as an example in this subsection. To predict the AVF in contention, we train a BRT model with the inputs being the program's AVF without resource contention and a group of occupancies measured under contention from the following structures: ROB, LSQ, IQ, RF, I/D TLBs, L1 I/D caches, and L2 cache. Figure 4-6 demonstrates the input variable importance derived from the ROB AVF predictor. We can see that the ROB AVF largely depends on the thread's register file occupancy. Register file plays an important role in the ROB AVF because the physical registers available for each thread are limited in a multi-way SMT processor. The processing of the instructions in the pipeline is thus more sensitive to the register file usage, and so is the AVF. In addition, we found that the ROB AVF of a program shows strong variation when it competes with different programs, but is consistently lower than that when the program runs alone (as exemplified in Figure 4-1). This is because the shared resource contention reduces the thread's register file occupancy to different extents in different benchmark combinations.

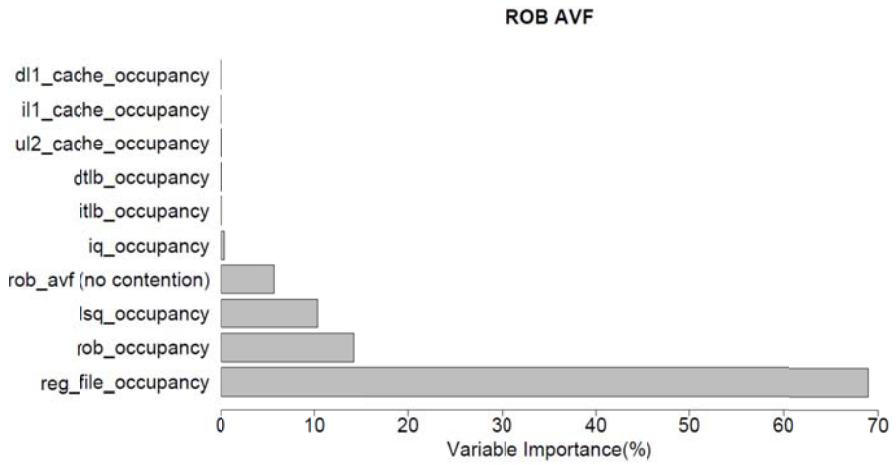

Figure 4-6. Input variable importance of ROB AVF in SMT processors

4.3.3 Impact on SMT Shared Structures

The other three SMT structures (i.e. IQ, Functional Units, and RF) in analysis are shared among different threads. For example, the instructions from both threads co-exist in the IQ of a two-way SMT processor, simultaneously contributing to the IQ AVF. Consequently, we need to provide the predictor with inputs from both threads during training. In this subsection, each of the two programs in a benchmark combination is simulated for 100 million instructions. We suffix the input variables from the faster thread (i.e. the one finishes earlier) with "0", and those from the slower thread with "1". For instance, "iq_occupancy_0" refers to the average IQ occupancy rate by the faster thread during simulation. As an example, Figure 4-7 shows the first ten most important inputs for register file AVF. As can be seen, both threads simultaneously affect the soft error vulnerability of a shared structure. Interestingly, for address-based structures such as IQ (figure not shown), the AVF still mainly depends on its occupancies; but for Functional Units and RF (Figure 4-7), the vulnerabilities when the two threads execute independently without contention become the most important variables. This actually indicates the necessity of our two-level model since it internally quantifies these important variables.

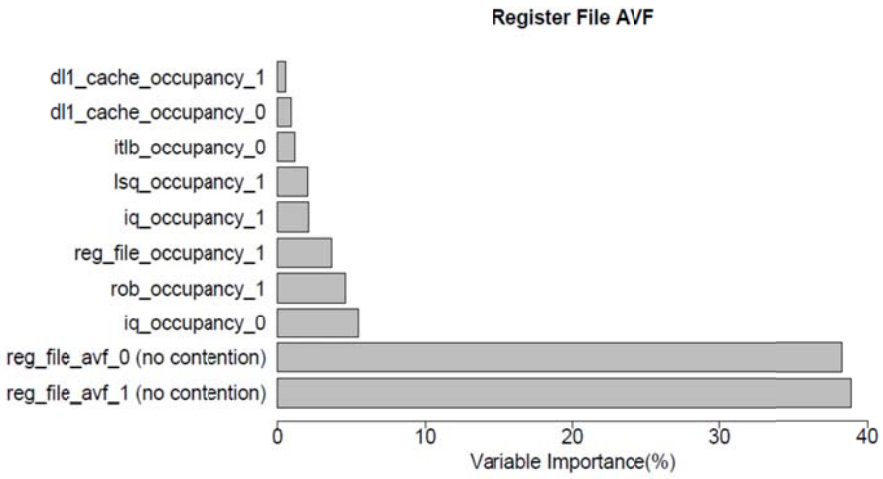

Figure 4-7. Input variable importance of Register File AVF in SMT processors

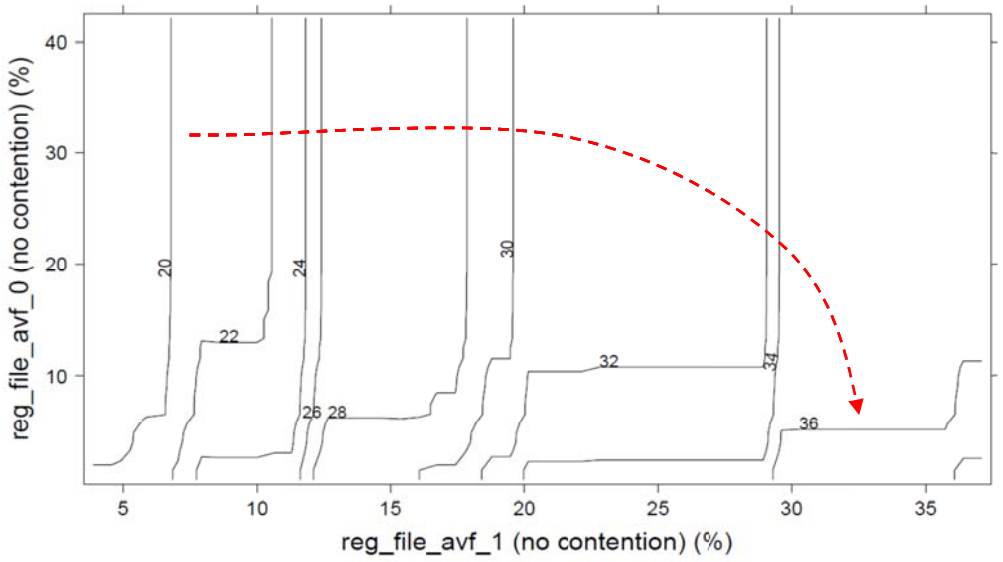

Figure 4-8. Partial dependence plot of Register File AVF on the two most important input variables in SMT processors

Figure 4-8 depicts the partial dependence of the register file AVF on the two contention-free AVFs. We can see that the RF AVF increases with respect to the increase in the slower thread's AVF and the decrease of the faster thread's (see the dashed curve). This indicates that the RF AVF would increase if one thread dominates the register file. In this case, more dependencies

exist between registers (since they are from the same thread), and the register file thereby contains more ACE bits that are vulnerable to soft errors. Furthermore, different from SMT private structures, the AVF of a shared structure can be higher or lower than the corresponding contention-free AVF (e.g. IQ AVF in Figure 4-1). The shared structure accommodates both threads, so there is always a certain portion of the structure that demonstrates probably completely different AVF behavior than that when running with one thread.

4.3.4 Impact on CMPs

In our dual-core CMP architecture, the two cores compete with each other via the shared L2 cache. Therefore, all the five processor structures in consideration are private to each core. We generally follow the same approach as above to train the predictors. The core AVF is the ration between the entire core's ACE bits and the core size. It can be calculated by summing up the individual components' AVFs weighted by the corresponding structures' sizes. We focus on the core AVF for CMP's study.

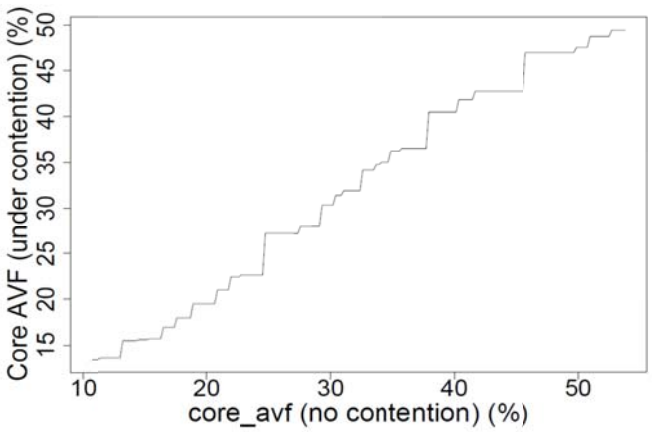

Figure 4-9. The partial dependence of CMP core AVF on its most important input

From our experiments, we found that the AVF of a CMP core running a certain benchmark varies in a very small range when different benchmarks execute on the other core. In other

words, the shared cache contention has relatively little impact on a CMP's core soft error vulnerability behaviors. This is in contrast with the situation in an SMT processor where the AVF shows significant variation. Figure 4-9 verifies the above finding by showing that the core AVF under contention is highly correlated to the core AVF without contention.

4.4 Two-Level AVF Prediction for Multi-Programmed Workloads

The model obtained in Section 4.3 assumes the awareness of the program's AVF on a single-threaded processor without contention, which can be predicted from performance measurements and configuration parameters on the single-threaded processor at the first level. This section first validates the first-level model, and then combines the two levels to demonstrate accurate predictions.

4.4.1 Single-Threaded Universal Model Validation

We tuned several important parameters to form a large processor design space (Table 4-2) at the first level. The values used in the baseline configuration (i.e. the one based on which we trained the second level model above) are shown in bold. The design space size is 1,244,160, from which we randomly and uniformly simulate 400 points for each of the benchmarks (see Table 4-1). After simulation, a few important but easily measured performance metrics are collected, including IPC, DL1/L2 cache miss rates, and the occupancies of ROB, LSQ, IQ and Register File. These performance measurements, along with the corresponding configuration parameters, are used to train the first level model. The simulation results of 300 configurations from each of the 30 training benchmarks (the 3rd column of Table 4-1) are used as training examples. The trained model is then tested with the other 100 configurations of the training benchmarks and all 400 configurations of the test benchmarks.

Table 4-2. Processor configuration design space composed of parameters P_1 to P_9. The values shown in bold are used in our baseline setting (Table 4-1).

	Parameter	Selected Values	# Options
P_1	Processor width	2, **4**, 8	5
	# Integer ALUs / # FP ALUs	1/1-associated with processor width 2 2/2, **4/4**-associated with processor width 4 4/4, 8/8-associated with processor width 8	
P_2	ROB size	72, 84, **96**, 108, 120, 132, 144, 156, 168	9
P_3	LSQ size	24, **32**, 40, 48, 56, 64	6
P_4	IQ size	32, 40, 48, 56, **64**, 72	6
P_5	L1 I/D cache size	**16**, 32, 64, 128 KB (64B block, 4-way)	4
	L1 cache latency	**1**, 2, 3, 4 cycles (vary with L1 cache size)	
P_6	L2 cache size	**512**, 1024, 2048, 4096 KB (64B block)	4
	L2 cache latency	**12**, 14, 16, 18 cycles (vary with L2 cache size)	
P_7	L2 cache assoc.	4, **8**	2
P_8	Branch predictor	bimod(1024), bimod(2048), **gshare(10-bit L1 width, 1024 L2 entries)**, gshare(11-bit L1 width, 2048 L2 entries), combined(1024), combined(2048)	6
P_9	BTB	**512/4**, 512/8, 1024/4, 1024/8	4

Because of the space limit, we only show prediction results for the core AVF in Figure 4-10. We use percentage error (i.e. |predicted value – true value| / true value * 100%) to report the prediction accuracy. Figure 4-10 is a boxplot showing the distribution of percentage errors for different benchmarks. In a boxplot, the upper and lower boundaries of the central gray box correspond to the upper and lower quartiles of all the errors; the highlighted horizontal line within the box is at the median; the vertical dotted line drawn from the box boundaries extend to the border lines for outliers. The top border line is Q3+1.5*IQR and the bottom one is Q1-1.5*IQR, where Q1 and Q3 are the first and third quartiles and IQR is the inter-quartile range Q3-Q1. Any observation outside that range is considered as an outlier, and denoted by a circle. From this figure we can see that the BRT-based universal model is very accurate for all benchmarks, achieving a 2.94% median percentage error on average. Note that here only one single model is constructed and

tested for all benchmarks. This is much more efficient than the application-specific model training and prediction approach adopted in traditional design space studies.

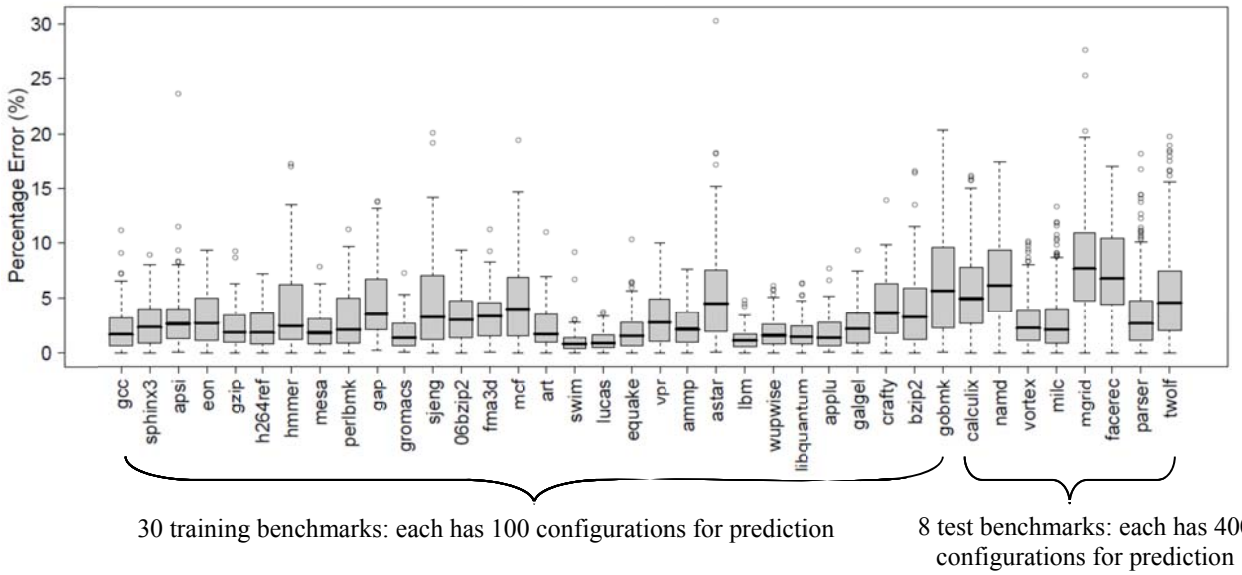

Figure 4-10. Prediction accuracy (core AVF) of the first level model. This model predicts 100 configurations for each of the 30 benchmarks in the training set (left) and 400 configurations for each of the 8 benchmarks in the test set (right).

4.4.2 Combing the Two Levels

Upon obtaining the models at both levels, we are able to combine them to perform the entire prediction. Figure 4-11 demonstrates the prediction accuracy in terms of percentage errors from the two-level prediction, which achieves high accuracy with median percentage errors of 2.64, 4.80, 3.76, 5.21, 11.56, and 14.03 percent for the five structures in SMT processors and the core AVF of CMPs, respectively. Note that the simulation results of the 8 test benchmarks (the 4th column of Table 4-1) are not used in model training at either level, validating the applicability of our predictive scheme to unseen programs. Besides, the two-level predictive model can work for both homogeneous and heterogeneous multi-threaded processors as long as the contention-related performance metrics can be measured. In reality, this can be easily satisfied since hard-

ware performance counters for key processor structures are already implemented in most commercial processors.

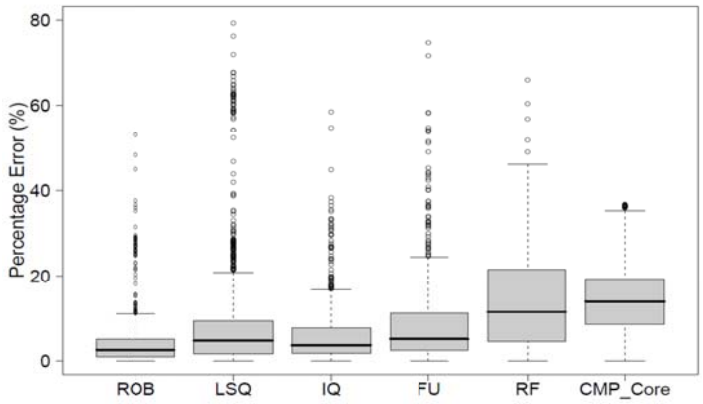

Figure 4-11. Prediction accuracy from two-level prediction

4.4.3 Model Scalability

Our predictive mechanism is scalable to more than two threads. Additional training to the predictor may be needed for some structures. To discuss the scalability issue, we further performed a set of 4-threaded (on four-way SMT and quad-core CMP) and 8-threaded (on eight-way SMT and eight-core CMP) multi-programmed simulations. For 4-threaded workloads, we evaluate the 70 combinations generated from the 8 test benchmarks (i.e. $C_8^4 = 70$); for 8-threaded workloads, we randomly choose 45 benchmark combinations from Table 4-1. In addition to more hardware threads, the physical register file size in four-way SMT processors is doubled compared to two-way ones; while in eight-way SMT, more parameters are reasonably enlarged.

As an example, Figure 4-12 shows the prediction results for ROB and LSQ AVFs in SMT processors, and the core AVF in CMP. For each of these three structures, three predictors are

trained: P1 is the one from Section 3 (i.e. only trained with 2-threaded examples); P2 is trained with both 2-threaded and 4-threaded examples (we put 25% 4-threaded simulation results in the training set); P3 is trained with 2-threaded, 4-threaded and 8-threaded examples (similarly, 25% 8-threaded simulation results are used in training). We evaluate the 4-threaded test examples with P1 and P2, and the 8-threaded test examples with all three predictors, respectively. For instance, P2(4) indicates the prediction accuracy for 4-threaded test examples using P2. As a result, there are 5 bars shown for each structure in this figure.

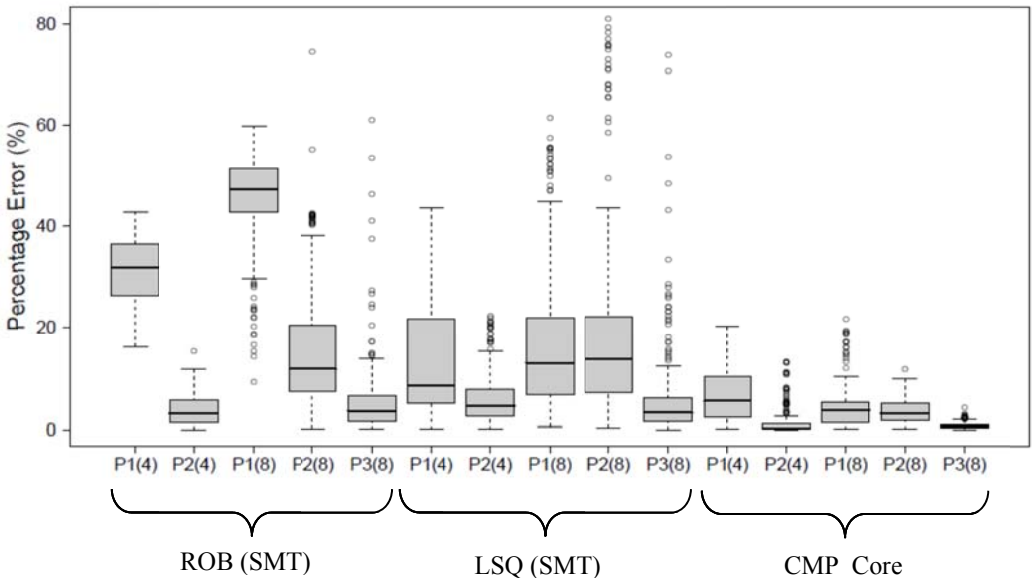

Figure 4-12. Prediction accuracy when scaling to 4 and 8 threads

For the ROB AVF, we can observe a big improvement in the prediction accuracy using the predictors with additional training (i.e. P2 and P3). Figure 4-6 illustrated that register file occupancy significantly affects ROB AVF in 2-threaded case. However, in 4-threaded SMT, the register file has been enlarged in our simulations, and is no longer the bottleneck. Therefore, ROB occupancy becomes the most influential factor. P1 didn't capture this behavior very well, thus performing poorly in 4-threaded and 8-threaded predictions. In contrast, in the case of CMP core

AVF, even P1 can make very accurate predictions when scaling to more threads. This is because the correlation between AVF and input variables does not change drastically for CMP with an increasing number of cores. To summarize, the predictors may need more training when scaling to more threads, but the proposed approach itself is scalable.

4.5 Extension to Multi-Threaded Workloads

Multi-threaded workloads (e.g. SPLASH2 [51]) explore thread-level parallelism. The multiple threads generated from the same program not only show contention in pipeline structures, but also show constructive behavior in memory hierarchy due to data sharing. For instance, a load miss from one thread can be avoided if the target data is recently accessed by another thread sharing the same cache. The impact of data sharing on the AVF of multi-threaded workloads is still largely unexplored. By varying the number of cores and application threads, Soundararajan et al. [42] concluded that the configurations optimizing soft error reliability of different multi-threaded applications are not straightforward. Consequently, an efficient approach to obtain the AVF of multi-threaded workloads is useful and necessary. In this section, we predict multi-threaded workloads' AVF from processor structure occupancies and data cache coherency states.

All experiments in this section are run using the M5 simulator [2] which is capable of simulating multi-threaded benchmarks. We implemented the AVF measurements for ROB, Load Queue, Store Queue and Issue Queue for multi-cores with Alpha 21264-like CPUs. 8 benchmarks (*cholesky*, *fft*, *radix*, *barnes*, *ocean.contiguous*, *ocean.noncontiguous*, *water-nsquared*, and *water-spatial*) in SPLASH2 suite are evaluated, each being measured with 2 threads, 4 threads, and 8 threads enabled on dual-core, quad-core, and eight-core processors, respectively. All cores in our CMP model have private L1 I/D caches and share a unified L2 cache. The data coheren-

cies among different L1 caches are maintained using a MOESI snooping protocol. Moreover, we dump the measurements every 500K cycles (called a phase) after program initialization, and collect about 100 phases for each simulation. We found that the AVFs demonstrate strong phase variation, but are very similar on different cores in the same phase. Therefore, we use the measurements from core 0 to represent the system in the following predictor training and testing.

In addition to structure occupancies and cache misses (as used in the above two-level predictor), we also include data cache coherency states as part of the inputs to characterize the data sharing among different threads. Specifically, the percentage of data cache blocks that are in each of the five states (i.e. "MOESI") is calculated. To our knowledge, this is the first work discussing the impact of cache coherency on soft error vulnerability. We train the predictor using the 4-thread phases of 6 benchmarks, and test it for the other two. Furthermore, this predictor is validated with the 2-thread and 8-thread phases of all 8 benchmarks (except a few 8-thread simulations not runnable under M5). The prediction results are shown in Figure 4-13. We can see that most predictions have median percentage errors lower than 10%. This demonstrates the predictor's effectiveness across workloads (the leftmost two bars in Figure 4-13), across architectures (predictions for 2-thread and 8-thread simulations), and across phases (prediction for different phases). Figure 4-14 quantifies the input variable importance from this predictor. As can be seen, while IQ occupancy is the most influential factor to the core AVF for multi-threaded programs, three coherency states (dcache.MODIFIED, dache.OWNED, and dcache.SHARED) also appear among the top 10 important factors.

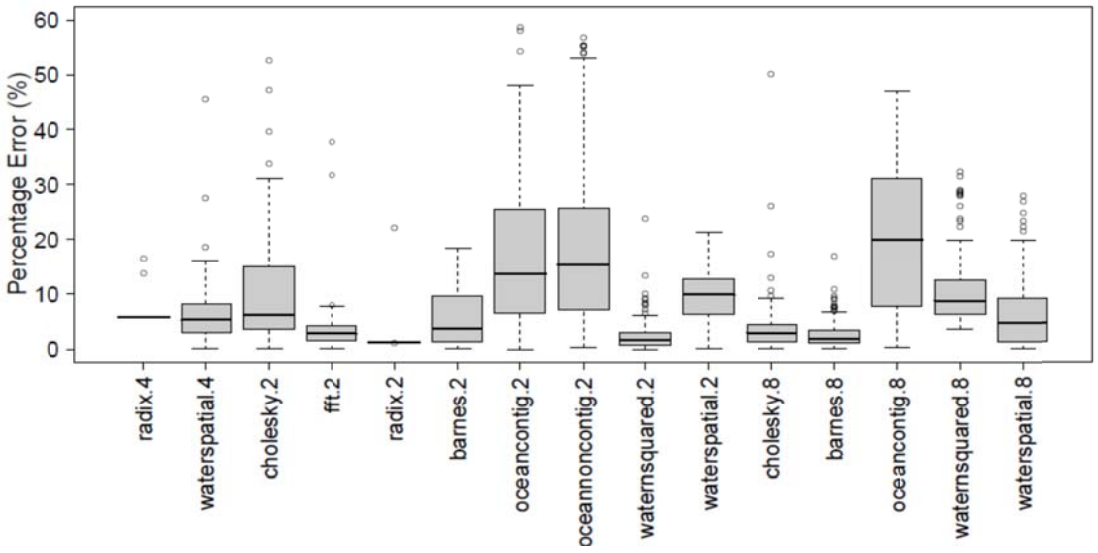

Figure 4-13. Prediction accuracy of SPLASH2 applications. The number at the end of an application name indicates the number of threads enabled.

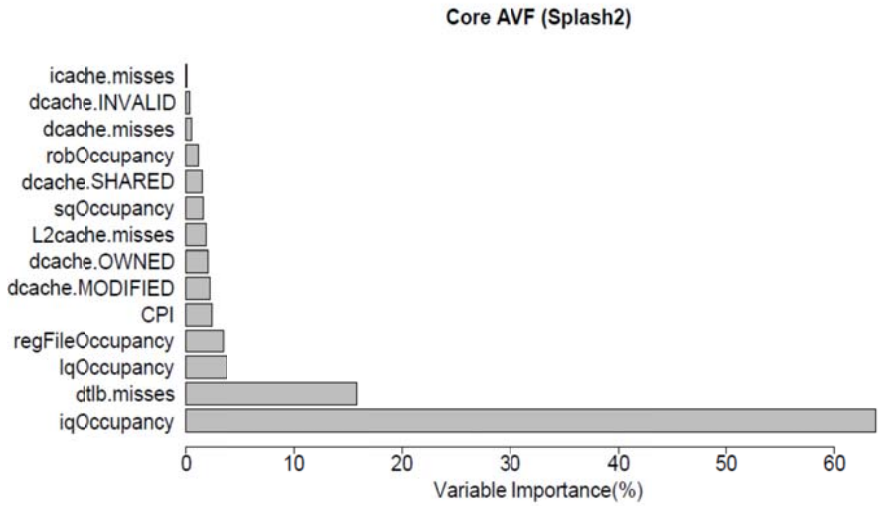

Figure 4-14. Input variable importance of the predictor for multi-threaded applications

4.6 Case Study: Soft Error Resilient Thread-to-Core Scheduling

This section presents a case study that identifies the optimal solution for soft error resilient thread-to-core scheduling on a Chip Multi-Threaded (CMT) processor [43]. A CMT processor

provides a combination of SMT and CMP. Programs co-scheduled on the same core (via SMT) compete for the shared pipeline resources, while different cores compete for the shared memory hierarchy. Therefore, different thread-to-core schedules may result in completely different behaviors in performance, power, and also AVF. Suppose each of the n cores in a CMT processor can support k simultaneous hardware threads, our goal is to find the optimal assignment of n*k programs (to different cores of this CMT processor) that minimizes the overall AVF.

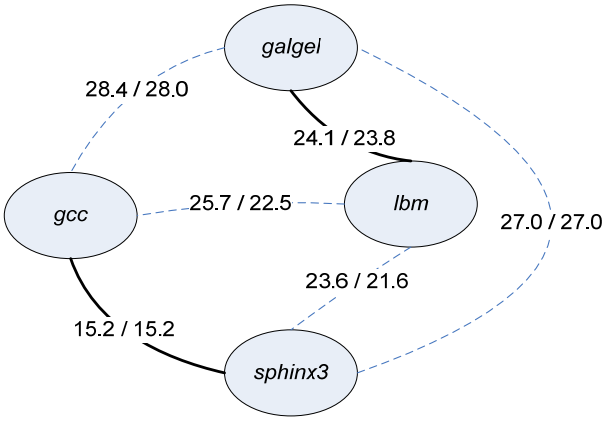

Figure 4-15. An example of identifying the optimal thread-to-core scheduling on a 2-way 2-core CMT processor. The optimal scheduling is shown in solid lines.

Figure 4-15 is a graphic representation of the problem when n = 2 and k = 2. In this completely connected graph, vertices represent the programs to be scheduled, and the weight of the edge between two vertices is given by the core AVF value when the two programs are co-scheduled on the same core. Therefore, finding the optimal program-to-core assignment (when k = 2) that minimizes the system AVF is identical to solving the *minimum-weight perfect matching problem*[5] in this graph. Jiang et al. [21] proposed a polynomial-time algorithm to solve this problem when k equals 2 (n can be larger than 2). They also proved that this problem becomes NP-complete when k is greater than 2.

[5] A matching in a graph is a set of edges without common vertices. A perfect matching is a matching that covers all vertices of the graph.

Consequently, as long as all the core AVF values (i.e. the weights of all the edges) are known when different combinations of programs are co-scheduled on the same core, we can easily identify the optimal thread-to-core assignment using Jiang et al.'s methodology [21]. The two-level AVF prediction proposed in this chapter provides an efficient approach to obtain these AVF values. In Figure 4-15, the two values shown on each edge are the measured/predicted core AVF when the corresponding two programs execute on the same core. Apparently, both measured and predicted values make the same correct decision: in this example, the optimal schedule is the assignment of (*gcc*, *sphinx3*) on one core and (*galgel*, *lbm*) on the other core. We can see that a suboptimal schedule, e.g. (*gcc*, *lbm*) and (*galgel*, *sphinx3*), will enlarge the system AVF by over 34% in this example.

Finally, to apply the above scheduling approach during program runtime, one can utilize Snavely et al.'s "Sample-Optimize-Symbios" (SOS) job scheduler [40]. SOS first runs each possible schedule for a short phase (i.e. "Sample"), selects the schedule with the highest goodness level (i.e. "Optimize"), and then runs the selected schedule for a number of intervals (i.e. "Symbios"). SOS periodically repeats the above procedure to choose the optimal schedule adapting to program runtime behavior. Figure 4-16 shows the runtime AVF behaviors of different schedules constructed from the previous four benchmarks. When SOS is applied, it simulates each of the three schedules for one interval, during which it collects the performance inputs and predicts the AVF for current schedule. After the sample phase, SOS sticks to the schedule with the lowest AVF for 10 intervals, and then reenters sample phase. We can see from this figure that SOS is very effective in identifying runtime optimal schedules.

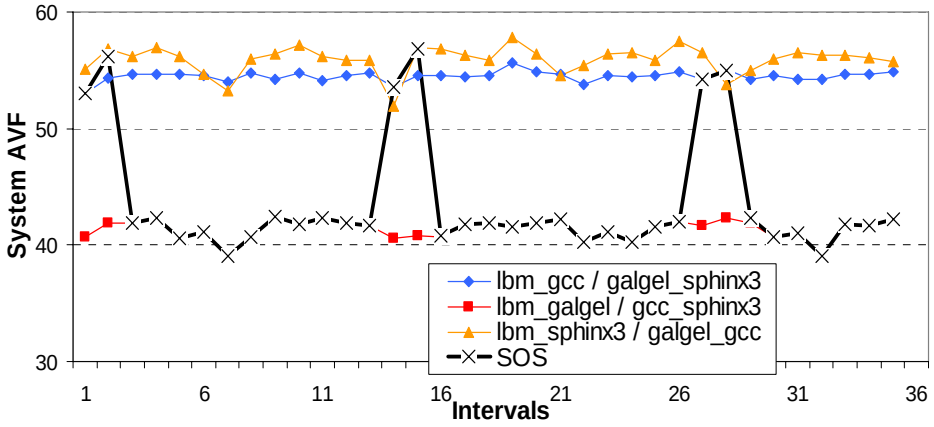

Figure 4-16. The runtime AVF behavior of different schedules. The SOS curve indicates the online AVF variation when "Sample-Optimize-Symbios" job scheduler is utilized.

4.7 Related Work

Recently, Sridharan et al. proposed Program Vulnerability Factor [44] and Hardware Vulnerability Factor [45] to describe architecture-independent program vulnerability and software-independent hardware vulnerability to soft errors. Regarding the studies on the inter-thread contention on multi-core architectures, Lee et al. [26] proposed a composable performance regression (CPR) scheme to predict the performance of a benchmark combination on a multiprocessor from configuration parameters; Chandra et al. [7] predicted the impact of contention on the shared cache using three performance models. Our work differs from theirs in the followings: (1) we study and predict the soft error vulnerability of processor structures; (2) we also evaluate the SMT structures where the inter-thread contention shows more significant impact; (3) the statistical technique used in our work provides useful model interpretations.

4.8 Conclusions

In this chapter, we propose a two-level predictive mechanism to accurately predict the soft error vulnerability of multi-threaded processors with resource sharing. The first level model cor-

relates the AVF in a contention-free environment with important performance metrics and the underlying processor configuration. The second level model takes as inputs the output of the first level model and a few performance measures under resource contention from multi-threaded processors. The proposed prediction is universal for different multi-program combinations and multi-threaded applications running on multi-threaded processors. Collectively, this work provides useful mechanisms and guidelines for soft error resilient multi-threaded processor design.

CHAPTER 5. UNIVERSAL RULES GUIDED DESIGN PARAMETER SELECTION

5.1 Summary

The previous two chapters address AVF prediction from performance measurements which can only be obtained when a processor exists. On the contrary, this chapter correlates AVF with design parameters, improving the soft error reliability at pre-silicon stage, i.e. before the processor is manufactured. We utilize PRIM mechanism to identify the design configurations that are inherently reliable to soft errors from a large design space.

The AVF is a combinational behavior of hardware and software [44][45]. A raw soft error can be masked in either the hardware system (e.g. idle bits in processor structures) or the program in execution (e.g. dynamically dead instructions). Consequently, a system's AVF value can span a wide range when the same program executes on different processor configurations (whose parameters are from a design space), while different programs may demonstrate completely different AVF behaviors on the same processor.

The situation is even more complicated in a multiprocessor where different cores share a low level cache. The AVF of one core may be affected by the other cores through the contention from the shared cache [53]. Figure 5-1 shows how the system AVF of certain configurations (cfg1 to cfg5) would vary when different workloads (multi-threaded benchmarks from SPLASH2 suite [51]) execute on a multi-core processor. The number at the end of a benchmark's name indicates the number of threads generated for that benchmark. cfg1 to cfg5 are chosen from a huge multiprocessor design space from which 1,000 design points are randomly sam-

pled and simulated for each workload in analysis. Only multiprocessors with homogeneous cores are considered here. The 1,000 sampled configurations are ranked in each workload in terms of their AVF values, with the lowest value as rank 1 and the largest value as rank 1000. We can see that, for the same configuration, its AVF rank could be significantly different across different programs or the same program with different numbers of threads (i.e. FFT). For cfg1-3, there exist one or more workloads suffering from significant reliability degradation, i.e. a high AVF rank. A reliable processor design would favor configurations (such as cfg4 and cfg5 in this example) whose AVF ranks are consistently low in different applications. However, the difficulty is how to quantitatively and efficiently identify such designs from a huge design space.

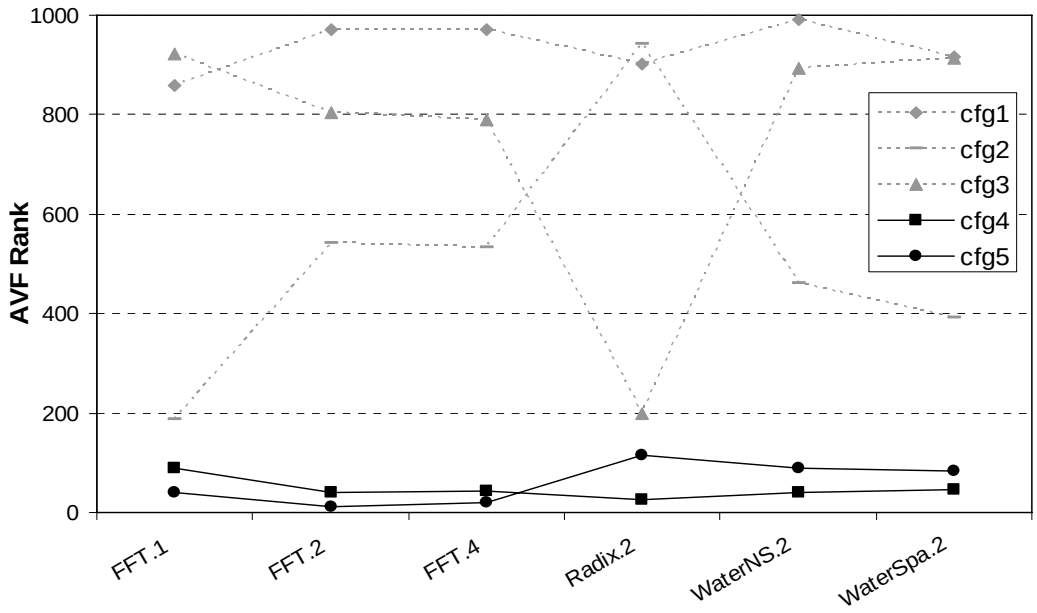

Figure 5-1. The AVF rank of a certain configuration varies significantly across different multi-threaded workloads. A lower rank indicates a lower AVF value that is favored in a reliable processor.

In this work, we propose an effective approach to identify the configurations that have consistently low AVF values from a huge design space. Those identified configurations are inherently reliable to soft errors. Specifically, we characterize the design space using *Patient Rule Induc-*

tion Method (PRIM) [17] to generate a set of selective rules on key design parameters. Applying these rules on the design space effectively identifies the subregion of the design space within which the output variable is considerably smaller (i.e. "valley seeking") than its average value over the entire design space. Therefore, the design configurations selected by the generated rules are inherently resilient to potential soft errors. This technique provides computer architects with useful guidelines to design reliable processors at pre-silicon stage.

More importantly, we are able to apply PRIM skillfully to derive a method that can generate "universal" rules effective across different applications. This cross-program capability is extremely useful in the era of multi-threading since the number of multi-threaded programs boosts quickly nowadays. For instance, the number of multi-programmed workloads increases superlinearly with the increases of the number of threads and the number of benchmarks in consideration. Traditional application-specific design space studies lack the scalability to multiprocessors due to the intractable training costs required by the greatly enlarged workload set. In contrast, our universal rules guided design parameter selection can provide workload-independent guidelines which are also validated to be effective for unseen applications not used in training.

Note, however, that by "universal" we don't refer to rules working for ALL programs (which may not even exist); instead, we manage to identify the rules with cross-program effectiveness for SPEC CPU and SPLASH2 benchmark suites used in this work. Nevertheless, these commonly used benchmarks are good representatives of real-world applications. Finally, the proposed approach is inherently generic, so it can be applied to optimize other processor design metrics, such as power and performance. A case study performed in this chapter utilizes the universal design parameter selection method to achieve a multi-objective optimization of reliability, performance, and power for multiprocessors.

In summary, the main contributions of this chapter are as follows:

- **Design parameter selection for reliable processors:** We propose to use an advanced rule search strategy (PRIM) to extract the design space subregion showing lowest AVF values. By using this technique, we quantitatively demonstrate that (1) minimizing the AVF for different processor structures may degrade or improve the performance; (2) reducing the AVF of a single structure may increase the AVF of others in the processor. This addresses the demand for a holistic reliability optimization.
- **Universal rules generation and validation:** We propose a generic approach capable of generating a set of "universal rules" that achieves the optimization of the output variable across different programs in execution. The effectiveness of the universal rule set is further validated using a well-developed statistical method (bootstrapping [15]) on programs that are not used in training.
- **Balancing reliability, performance and power for multiprocessors:** We perform a study using the proposed universal modeling scheme to simultaneously balance multiple design metrics for multiprocessors. We quantitatively identify proper trade-offs of reliability, performance and power for a multi-core processor running multi-threaded workloads.

5.2 Design Parameter Selection for Uniprocessors

5.2.1 Experimental Setup

We implement the AVF calculations in an extended version of SimpleScalar3.0 [39] to simulate a detailed out-of-order multistage superscalar processor. Our simulation framework measures the AVF of major microarchitectural components in a uniprocessor, including ROB,

LSQ, Functional Unit, and Register File. The processor overall AVF is the ratio between the number of ACE bits of the entire processor and the processor size in bits. It can be calculated as the summation of different structure's AVFs weighted by the corresponding structures' sizes.

We use a mixed set of benchmarks from SPEC CPU 2000 and 2006 suits. As listed in Table 5-1, 24 of them form the training set in generating universal rules, while the rest 12 are used for test. In order to have a complete evaluation, the entire SPEC CPU 2000 suite (except *sixtrack*) is included. For the other SPEC CPU benchmarks not used in this work, we are not able to compile them into Alpha binaries runnable in SimpleScalar. Each benchmark is simulated for 100 million instructions in details after being fast forwarded to a representative phase derived from SimPoint Toolkit [38]. The AVF values along with performance data are outputted at the end of each simulation.

For this study on uniprocessors, we construct a design space composed of eight configuration parameters ($\mathbf{P_1}$ to $\mathbf{P_8}$ in Table 5-2). Note that the ROB and LSQ sizes are configured in a fine-grained manner because our scheme intends to quantify the value ranges of the parameters in the optimal configurations. Given the possible values of each parameter, the total number of points in the design space is 473,088. For each benchmark, we simulate 2,000 configurations randomly and uniformly sampled from the entire design space.

Table 5-1. SPEC benchmarks used in the uniprocessor study.

Train (24)	Test (12)
applu, apsi, art, bzip2, crafty, equake, fma3d, gcc, mcf, mesa, perlbmk, twolf, vortex, astar, 06bzip2, gobmk, hmmer, libquantum, lbm, 06mcf, milc, namd, sjeng, sphinx3	*ammp, eon, facerec, galgel, gap, gzip, lucas, mgrid, parser, swim, vpr, wupwise*

Table 5-2. The uniprocessor design space is composed of parameters P_1 to P_8. Branch predictors are renamed to *BP1* to *BP8* for later use.

	Parameter	Selected Values	# Options
P_1	Processor width	2, 4, 8	6
	Fetch queue size	2, 4, 8 (vary with processor width)	
	# of Integer ALUs / # of FP ALUs	1/1, 2/1-associated with processor width 2 2/1, 2/2-associated with processor width 4 2/2, 4/4-associated with processor width 8	
P_2	ROB size	64, 70, 80, 90, 100, 110, 120, 130, 140, 150, 160	11
P_3	LSQ size	16, 24, 32, 40, 48, 56, 64	7
P_4	L1 I/D cache size (L1CS)	16, 32, 64, 128 KB (32B block, 2-way)	4
	L1 cache latency	1, 2, 3, 4 cycles (vary with L1 cache size)	
P_5	L2 cache size (L2CS)	512, 1024, 2048, 4096 KB (64B block)	4
	L2 cache latency	8, 12, 16, 20 cycles (vary with L2 cache size)	
P_6	L2 cache associativity (L2CA)	4, 8	2
P_7	Branch predictor (BP)	bimod/4096 (*BP1*), bimod/8192 (*BP2*), 2lev/1/4096 (*BP3*), 2lev/2/4096 (*BP4*), 2lev/4/4096 (*BP5*), 2lev/1/8192 (*BP6*), 2lev/2/8192 (*BP7*), 2lev/4/8192 (*BP8*)	8
P_8	BTB	1024/4, 2048/2, 1024/8, 2048/4	4

5.2.2 Application-Specific Design Parameter Selection

In this subsection, we directly apply the PRIM method and build a separate model for each benchmark to minimize its AVF. In other words, the derived rules are towards the optimization of the AVF for a specific application. For each benchmark, the rule sets minimizing the AVF of different structures are generated. From these results, we observe that minimizing the AVF of different structures have different impacts on the performance. Table 5-3 lists the results for a set of benchmarks. The rules for other benchmarks show similar behaviors, thus being omitted.

First, some rules tend to degrade the performance considerably when minimizing the AVF. Specifically, the rules for *mcf* to optimize the ROB AVF introduce restrictions on branch predictor selection. Figure 5-2 shows the variation of branch misprediction rate and the ROB AVF

when different branch predictors *BP1* to *BP8* are used in the configurations running *mcf*. The other parameters are the same for these configurations. Clearly, the ROB AVF varies contrarily with respect to the variation of branch misprediction rate. If we exclude *BP1* and *BP2* as the rules suggest, one can expect a significant performance loss. Similar observation can be made from the rules for minimizing the LSQ AVF. For example, in *gobmk*, the restriction on L1 cache size to have smaller values will result in a larger execution time.

Table 5-3. Rules for optimizing individual structure's AVF for a set of benchmarks. "*Width/ALUs*" is the combination of processor width, # of integer ALUs, and # of FP ALUs; '&' refers to 'AND'; '||' refers to 'OR'.

Bench-mark	ROB AVF	LSQ AVF	Functional Unit AVF	Register File AVF
mcf	(ROB > 130) & (LSQ < 64) & (BP!= *BP1*) & (BP!=*BP2*)	(LSQ > 48) & (L2CS < 4096) & (BP!=*BP1*) & (BP!=*BP2*)	Width/ALUs=8/4/4	(Width/ALUs!=2/1/1) & (LSQ > 24) & (BP=*BP1*||*BP2*)
applu	(ROB > 90) & (LSQ < 32)	(ROB < 110) & (LSQ > 40)	(Width/ALUs=4/2/2||8/2/2|| 8/4/4) & (LSQ < 32)	(Width/ALUs=4/2/2||8/2/2||8 /4/4) & (ROB > 80) & (LSQ > 24) & (L1CS > 16)
fma3d	(Width/ALUs=2/2/1) & (ROB > 90)	(Width/ALUs=2/2/1||4/2/1||8/ 4/4) & (LSQ > 40) & (L1CS < 128)	(Width/ALUs=2/1/1||8/4/4) & (L1CS < 64)	Width/ALUs=8/4/4
gobmk	(ROB>110) & (L1CS<128) & (L2CS > 512) & (BP!=*BP2*) & (BP!=*BP1*)	(LSQ > 40) & (L1CS < 64) & (BP!=*BP1*) & (BP!=*BP2*)	(Width/ALUs=2/2/1||8/4/4) & (L1CS < 64)	(Width/ALUs!=2/1/1) & (Width/ALUs!=2/2/1) & (L1CS > 32) & (L2CS > 512) & (BP!=*BP5*) & (BP!=*BP3*) & (BP!=*BP4*)
milc	(ROB>90) & (LSQ<32)	(ROB < 110) & (LSQ > 40)	Width/ALUs=8/4/4	(Width/ALUs!=2/1/1) & (ROB > 110) & (LSQ > 40)

In contrast, optimizing the Register File AVF simultaneously improves the performance. Therefore, the rules generated for minimizing the Register File AVF clearly select the designs achieving high performance, e.g. larger ROB or LSQ, larger caches, wider CPUs, or more accurate branch predictors. For example, the rules for *milc* favor a ROB size larger than 110. Figure 5-3 shows the RF AVF and performance variation trends under different ROB sizes. It is easy to see that a larger ROB size results in better performance and a more reliable Register File as well. More pipeline resources will improve performance, making instructions pass through the pipe-

line more quickly. This will shorten the write-read interval for a certain register, thus reducing ACE cycles and decreasing the register file AVF.

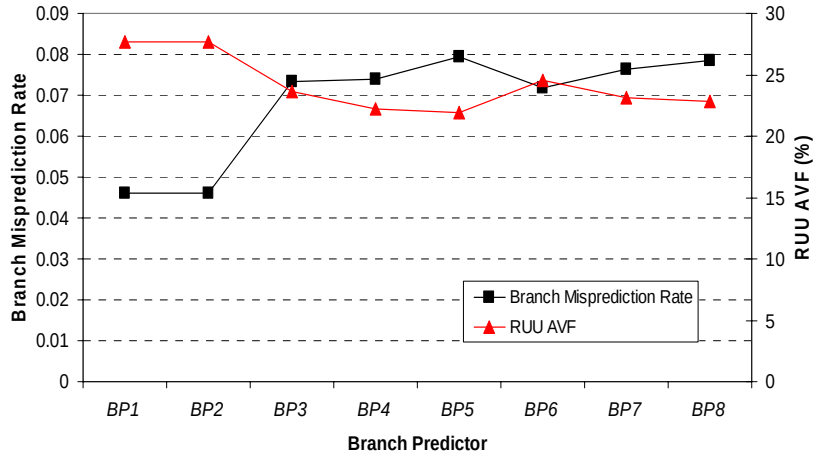

Figure 5-2. ROB AVF of *mcf* (SPEC 2000) varies with different branch predictors

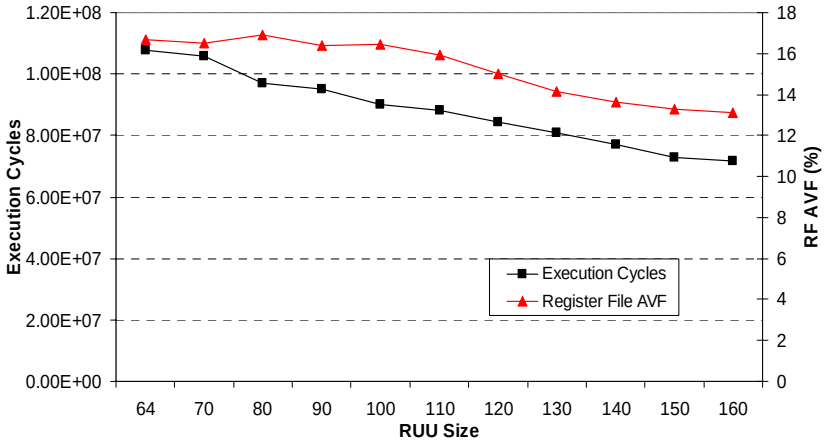

Figure 5-3. Register File AVF of *milc* (SPEC 2006) varies with different ROB sizes

The correlation between the AVF and performance is fuzzier in Functional Units. A wide processor usually incurs a low FU AVF because it has more ALUs which will execute the instructions more quickly (thus fewer ACE cycles); a narrow processor significantly degrades performance, but its FU AVF may be still low due to the inefficient usage of ALUs. This can be

verified from the rules for optimizing the FU AVF of many benchmarks. For instance, Figure 5-4 illustrates that in *fma3d* the execution cycles consistently decrease with increased processor width and number of ALUs, but the FU AVF increases initially and decreases later. Consequently, the rules for *fma3d* shown in Table 5-3 choose the two extreme settings (either widest or narrowest) in optimizing the FU AVF. If performance is taken into consideration, one should choose a wide processor.

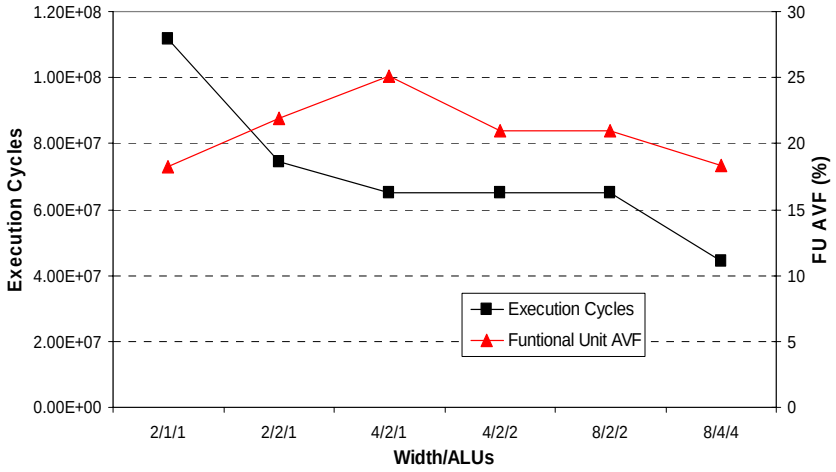

Figure 5-4. Functional Unit AVF of *fma3d* (SPEC 2000) varies with different combinations of processor width and # of ALUs

To summarize, minimizing the individual processor structure's AVF may degrade the performance (e.g. in ROB and LSQ), or improve the performance (e.g. in Register File), or result in either way (e.g. in Functional Unit).

5.2.3 Universal Rules Guided Design Parameter Selection

In Table 5-3, we can observe many contradictions between the rule sets for optimizing the AVFs of different structures. For example, *applu* requires a small LSQ size (<32) but a large one (>40) in optimizing the AVF for ROB and LSQ, respectively. Figure 5-5 illustrates this contradiction. We can see that the LSQ AVF decreases with the increase of LSQ size, but in the mean

time the ROB AVF quickly boosts to a very high value. Therefore, if the rules for optimizing the LSQ AVF are adopted in a processor design, ROB will become extremely vulnerable. Consequently, reducing the AVF of one processor structure may increase the AVF of others. When designing a reliable processor, one can easily make a mistake by transferring the soft error vulnerability to other parts of the processor, instead of really reducing it. Therefore, the overall AVF of the entire processor should be considered to achieve a holistically reliable solution.

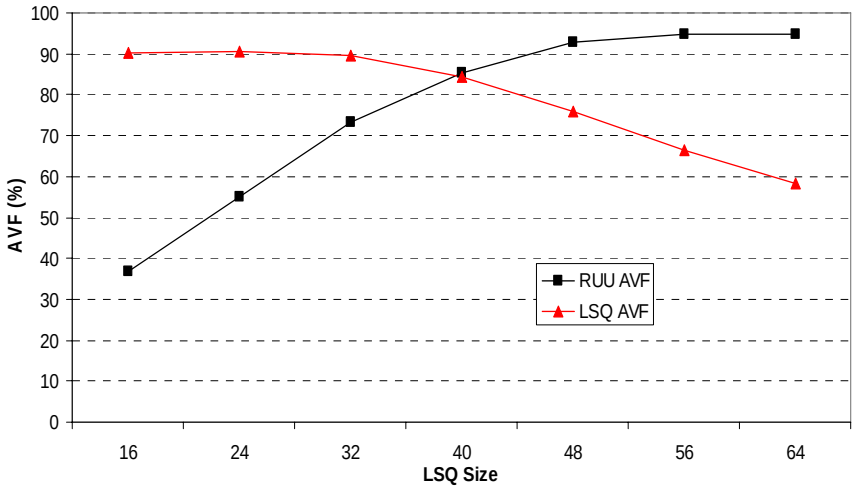

Figure 5-5. ROB and LSQ AVFs of *applu* (SPEC 2000) vary with different LSQ sizes

All the rule sets in Table 5-3 are generated specifically for a certain benchmark. In other words, PRIM is directly applied to the AVF measurements of that particular benchmark to summarize the rules. Consequently, the rule sets work well for their corresponding applications, but differ from each other. From Figure 5-1 and Table 5-3, we can see that there exists some consensus among different programs about what configurations are reliable. Therefore, it is possible to extract a "universal" rule set that works well across different programs.

However, directly combining the AVF measurements from different benchmarks for training is not feasible since the absolute AVF value ranges significantly differ in different bench-

marks. Instead, we propose to rank the configurations in each benchmark in terms of the AVF measurements. For the 2K configurations used in this work, the one with the lowest AVF value is ranked 1 while the one with the largest AVF is ranked 2000. If so, a certain configuration would have 24 different ranks for the 24 training benchmarks in Table 5-1, respectively. We use the average of these ranks as the output response to train PRIM models. The generated rule set contains the design points that are universally reliable for all training benchmarks. An alternative is to look at the maximum of the ranks, but the results generated in this way are more conservative. In practice, we found that minimizing the average of the cube of the ranks (i.e. mean($rank^3$)) is very effective in identifying the universal rules, as this tends to balance the ranks across different benchmarks. For example, suppose we have 5 benchmarks and need to compare two cases with ranks (2, 2, 2, 2, 2) and (1, 1, 1, 1, 6), respectively. If the average of ranks is used, the two cases are considered as the same; but if the cube of ranks is used, the first case is better than the second one.

By using the above approach, we are able to generate a universal rule set from the training benchmarks that optimizes the overall AVF of a uniprocessor. It is shown as Rule Set I in Figure 5-6. Rule Set I provides useful guidelines in designing a holistically and universally reliable processor. It favors a large ROB size because ROB has the largest contribution to the core AVF; other factors somehow degrade the performance, validating our previous observation that a contradiction exists between optimizing performance and some structures' AVF. The next subsection will test this rule set on other benchmarks to validate its effectiveness on unseen programs.

(Width/ALUs!=8/2/2) & (ROB>130) & (LSQ<24) &
(L1CS<128kB) & (BP!=*BP1*) & (BP!=*BP2*)

Figure 5-6. Rule Set I (Optimizing Uniprocessor AVF)

5.2.4 Universal Rules Validation

In this subsection, we apply Rule Set I on the 12 test benchmarks (see Table 5-1) to validate its effectiveness in identifying reliable design configurations. For each benchmark being tested, the validation consists of the following steps:

(1) Simulate 2,000 configurations randomly and uniformly sampled from the entire design space. These simulations are used to approximate the whole design space whose exhaustive simulation is intractable.

(2) Identify what configurations among the 2,000 ones are selected by Rule Set I. When Rule Set I was generated above, β was set to 2%. Therefore, there are approximately 40 points selected by this rule set.

(3) Identify in which part of the design space the points selected by Rule Set I are actually located.

The main difficulty of the above approach is in (3), because for each benchmark in the test set we intend to know where those configurations selected by the rule set are located in the entire design space (not just the sampled 2K configurations!). In other words, we intend to know what percentile (say p) of the design space that the values of these selected points are below. The p-percentile for the whole space indicates the value that is greater than p% of all the design points but less than the rest. In order to make inference based on the entire design space, we use the bootstrapping method [15]. Specifically, we first sample (with replacement) 1000 bootstrap samples for the 2000 configurations. Note that each bootstrap sample also contains 2000 design points. We then compute a confidence interval estimate of the p-percentile of the entire design space based on these samples. Specifically, for each bootstrap sample, we calculate its p-percentile. This gives us a total of 1000 values for p-percentiles (one for each bootstrap sample).

Among these 1000 values, we further calculate their 5-percentile (say W). By doing so, we have 95% confidence that the *p*-percentile of the entire design space is larger than or equal to W. Finally, we adjust the *p* value (by repeating the above steps) to have the derived W slightly larger than the largest value of the selected points. Therefore, the final determined *p* value is the percentile that all the selected points are below. This approach is conservative since the exact *p*-percentile of the entire design space could be much larger than W.

For each benchmark being tested, we first calculate the minimum, lower quartile, median, upper quartile, and maximum of the design points selected by the rule set; after that, for each of these five values, we calculate the corresponding percentile of the entire design space it is below (using the bootstrapping method). We use boxplot to demonstrate the validation results in Figure 5-7. In a boxplot, the upper and lower boundaries of the central gray box correspond to the upper and lower quartiles; the highlighted horizontal line within the box is at the median; the vertical dotted line drawn from the box boundaries extend to the minimum and maximum. The vertical axis shows the percentile of the entire design space that the selected points are below. For example, for *vpr*, the maximum of the points selected by Rule Set I corresponds to a value of 5% in the vertical axis, meaning that in this benchmark all selected points are within the top 5% optima of the entire design space. We can see that Rule Set I is very effective in finding the optima for all test benchmarks. On average, the design points quantified by Rule Set I achieve the top 10% optima of the entire design space. Again, as clarified in Section I, we don't intend to locate the design space subregion that is reliable independent of all programs, but demonstrate that the rules generated using our proposed methodology work well across SPEC CPU benchmarks. These rules would be effective for other programs outside SPEC provided that SPEC CPU benchmark suites well represent real-world applications.

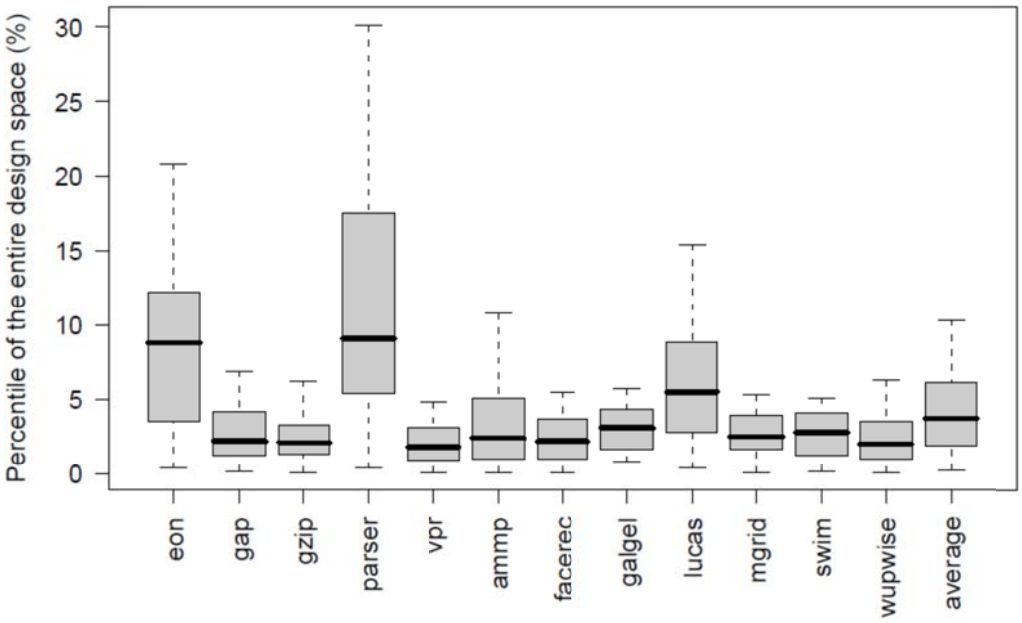

Figure 5-7. Validation of Rule Set I on the test benchmarks

5.3 Balancing Reliability, Performance and Power for Multiprocessors

We further extend our universal PRIM modeling scheme to multiprocessors, demonstrating that universally soft error resilient design configurations still exist for a homogeneous multi-core processor running multi-threaded workloads. By varying the number of cores and application threads, Soundararajan et al. [42] concluded that the configurations optimizing soft error reliability of different multi-threaded applications are not straightforward. Our proposed scheme can still quantify and validate the optimal subspace for multiprocessors. Furthermore, we perform a multi-objective optimization in this section that concurrently balances multiple design metrics (reliability, performance and power) for a multiprocessor.

5.3.1 Experimental Setup

All experiments in this section are run using the M5 simulator [2] capable of simulating multi-threaded benchmarks that have data sharing among threads. Consequently, the result of an instruction that is "dynamically dead" in one thread may be used in another thread, making it "vulnerable" as well. Therefore, in order to calculate the AVF for multi-threaded workloads, a system-wise post-commit analysis window needs to be maintained. The committed instructions from different threads are inserted into this unified window, and their types can be determined after reaching the other end of the window. The AVF can then be calculated from such information. We implement the AVF measurements for ROB, Load Queue, Store Queue and Issue Queue for multiprocessors with Alpha 21264-like CPUs.

Six benchmarks (*Cholesky, FFT, Radix, OceanContiguous, WaterNSquared,* and *WaterSpatial*) in SPLASH2 [51] suite are evaluated, each being measured with 1 thread, 2 threads, and 4 threads enabled on single-core, dual-core, and quad-core processors, respectively. All cores in our multiprocessor model have their private L1 I/D caches and share a unified L2 cache. The data coherencies among different L1 caches are maintained using a MOESI protocol. Multi-threaded workloads explore thread-level parallelism. The multiple threads running simultaneously show contention as well as constructive behaviors in the shared memory hierarchy. Therefore, the AVF, performance and power of one thread can be affected by its resource competitors.

M5 simulates Alpha 21264-like out-of-order CPUs, whose important parameters are tuned and form a new design space shown in Table 5-4. In this study, 1,000 configurations are randomly sampled from the multiprocessor design space and simulated for each benchmark. Note that a separate core (with the corresponding configuration from the design space) is created for each of the threads enabled in the simulated benchmark. The detailed simulation starts after the pro-

gram's sequential initialization, and stops when the fastest thread finishes a certain amount of instructions.

Table 5-4. The multiprocessor design space is composed of parameters M_1 to M_9. Only multiprocessors with homogeneous cores are considered. The entire space size is 1,458,000.

	Parameter	Selected Values	# Options
M_1	Processor width	2, 4, 8	6
	# of Integer ALUs / # of FP ALUs	1/1, 2/1-associated with processor width 2 2/2, 4/3-associated with processor width 4 4/3, 6/4-associated with processor width 8	
M_2	ROB size	72, 84, 96, 108, 120, 132, 144, 156, 168	9
M_3	LQ/SQ sizes	16, 20, 24, 28, 32	5
M_4	IQ size	32, 40, 48, 56, 64, 72	6
M_5	Phys. Int/FP reg. file sizes	100, 120, 140, 160, 180	5
M_6	BTB	1024, 2048, 4096	3
M_7	RAS	8, 12, 16	3
M_8	L1 I/D cache sizes	16, 32, 64, 128 KB (64B block, 2-way assoc.)	4
	L1 cache latency	1, 2, 3, 4 cycles (vary with L1 cache size)	
M_9	(Shared) L2 cache size	512, 1024, 2048, 4096, 8192 KB (64B block, 8-way assoc.)	5
	(Shared) L2 cache latency	10, 12, 14, 16, 18 cycles (vary with L2 cache size)	

5.3.2 Optimizing Individual Metrics

Before simultaneously balancing the three metrics, we separately optimize each of them first. The multiprocessor's soft error vulnerability can be characterized by its aggregated AVF: in our case, it's the average of all cores' AVFs because of the core homogeneity; the reciprocal of the system throughput, i.e. $1/\text{Throughput} = (\Sigma\ IPC_i)^{-1}$, where $0<=i<n$, is used to represent a n-core processor's performance; finally, the total power is the summation of all cores' power. Note that all three metrics favor a lower value.

We follow the same approach described in Section 5.2.3 to generate universal rule sets optimizing the three metrics, respectively. Specifically in this work, we put the 2-thread runs of 5

benchmarks (except *WaterSpatial*) in the training set, and validate the generated rules with 1-thread and 4-thread runs. In particular, *WaterSpatial* is chosen to have all configurations (including 1-thread, 2-thread, and 4-thread) in the test set, validating the generated model's effectiveness across different SPLASH2 benchmarks and different numbers of threads. These three rule sets (Rule Set II, III, IV) are listed in Table 5-5. Not surprisingly, a multiprocessor with wider CPUs and more pipeline resources usually demonstrates better performance, while a power-efficient design often selects parameters at lower end of the range. In contrast, Rule Set II which minimizes the processor AVF favors a large value in some structures (e.g. ROB) but a small value in some others (e.g. LSQ, IQ). The validation of these three rule sets are shown in Figure 5-8. We can see that most identified designs are within the top 20% optima.

Table 5-5. Universal rule sets for optimizing different metrics for multiprocessors

Rule Set II (Optimizing AVF)	(Width/ALUs=4/2/2||4/4/3||8/4/3||8/6/4) & (ROB>132) & (LSQ<32) & (IQ<48) & (L1CS>32kB) & (L2CS<2MB)
Rule Set III (Optimizing Throughput $^{-1}$)	(Width/ALUs=8/4/3||8/6/4) & (IQ>64)
Rule Set IV (Optimizing Power)	(Width/ALUs=8/6/4) & (ROB<156) & (16<LSQ<32) & (IQ<72) & (Phy. Reg. File<140) & (16kB<L1CS<128kB) & (L2CS !=1MB)
Rule Set V (Optimizing AVF $^{0.3}$ * Throughput $^{-0.4}$ * Power $^{0.3}$)	(Width/ALUs=8/4/3||8/6/4) & (ROB>132) & (LSQ < 32) & (IQ<56) & (BTB>1024) & (L1CS<128kB) & (L2CS != 4MB)
Rule Set VI (Optimizing AVF $^{0.2}$ * Throughput $^{-0.6}$ * Power $^{0.2}$)	(Width/ALUs=8/4/3||8/6/4) & (ROB>120) & (LSQ < 32) & (IQ<48) & (L1CS<128kB) & (L2CS != 8MB)

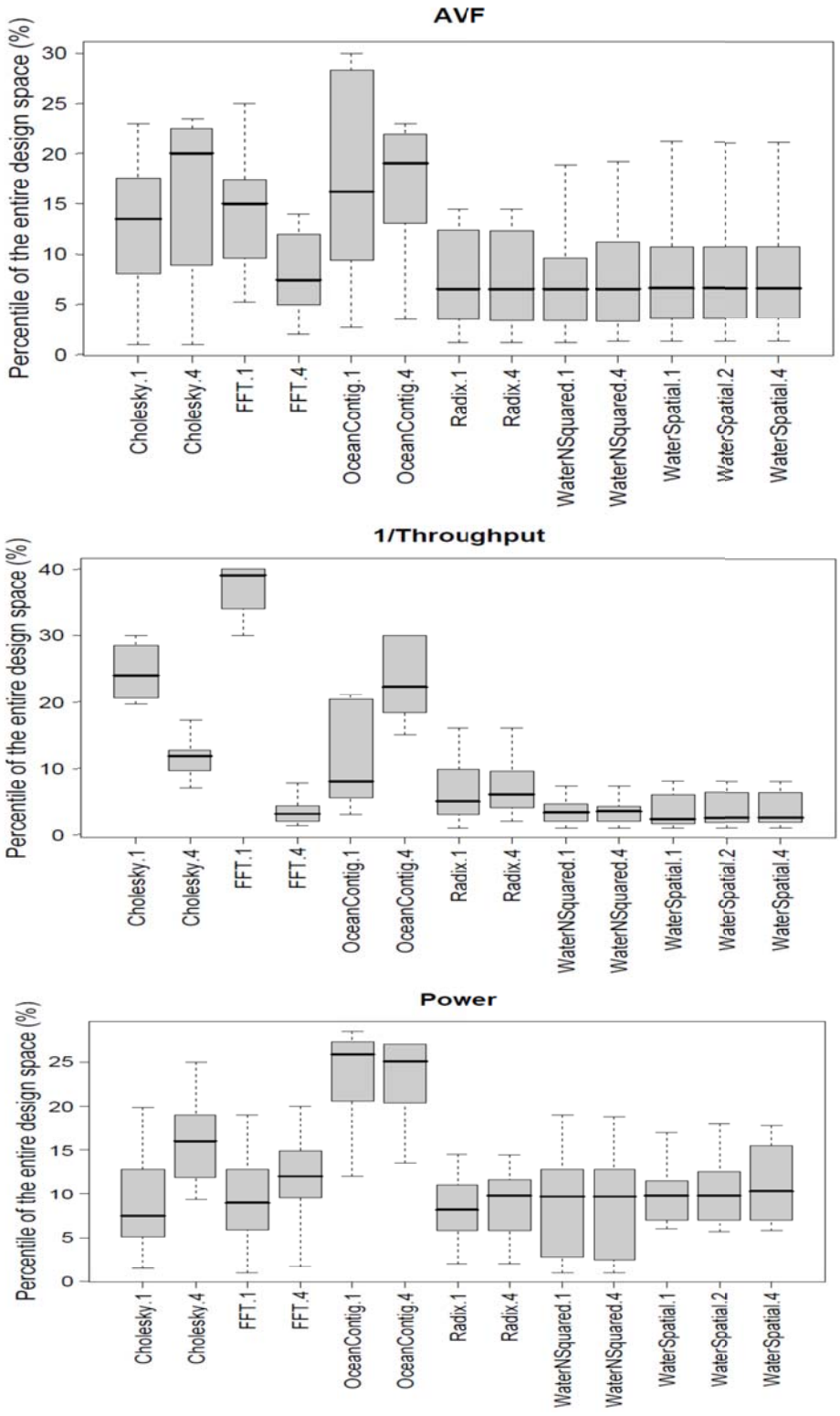

Figure 5-8. Validation of Rule Set II, III, IV on the test multi-threaded benchmarks. The number at the end of a benchmark's name indicates the number of threads.

5.3.3 Balancing Multiple Metrics

Simultaneously balancing the three metrics is actually a multi-objective optimization problem, requiring a reasonable objective function. We propose to minimize the function f in Figure 5-9 to achieve a good trade-off among different conflicting metrics.

$$f = AVF^{a} * (1/Throughput)^{b} * Power^{c}$$

where a, b, c >= 0, and a+b+c = 1

Figure 5-9. The objective function used in balancing multiple metrics

The exponentials a, b, and c are weight factors controlled by the designer. Formulating the objective function as above would result in an optimization process in proportional to the relative change of different metrics, ensuring more fairness than other objective functions such as normalized summation. Consequently, the designer can give more importance to a certain metric by enlarging its weight factor. The above discussions regarding Rule Set II, III, and IV are actually special cases where one of the three weight factors equals 1 and the other two equal 0. That said, Rule Set II – IV merely optimize a certain metric without taking the other two into account. Therefore, one can expect large degradations in the other two metrics for each of the three rule sets.

Figure 5-10 shows the comparison of different assignments of weight factors (a, b, c) in terms of AVF, performance and power. A separate rule set is generated for each weight factors assignment. Each column in this figure corresponds to the average response of the design points selected by the corresponding rule set, and this value is normalized to the case merely optimizing the response metric (i.e. one in Rule Set II, III, IV). For example, the AVF of the configurations selected by Rule Set III (weight factors (0, 1, 0)) demonstrates 64.5% degradation compared to those selected by Rule Set II which merely optimizes the AVF; in contrast, Rule Set II shows

36.6% degradation in performance than Rule Set III. Hence, none of the three rule sets (II, III, and IV) provides good balance of reliability, performance and power. This is in our expectation since they individually optimize only one of the three conflicting metrics. On the other hand, tuning the weight factors would result in better trade-offs among the metrics. (0.3, 0.4, 0.3) is a well-balanced assignment which shows in the figure 8.4%, 16.3%, and 1.3% degradations in AVF, performance and power, respectively. One can further enlarge the performance's weight factor to mitigate the performance loss. For instance, (0.2, 0.6, 0.2) is another assignment that decreases performance degradation to 9.4% but comes with an increased AVF degradation to 14.3%. The rule sets for these two assignments are also listed in Table 5-5 as Rule Set V and VI.

5.4 Related Work

For the correlation between the AVF and configuration parameters, Cho et al. [8] predicted the dynamics of power, CPI and the AVF using a combination of wavelets and neural networks. They also followed the same approach to predict the average soft error vulnerability and its tradeoff with performance [9]. Our work differs from theirs in that we provide simple but helpful guidelines to conduct reliable processor design. We also quantitatively analyze the effect of optimizing holistic reliability and identify the trade-off of reliability, performance, and power for multiprocessors.

A series of studies discussed design space exploration on performance and/or power [30][31]. Ipek et al. [22] predicted performance of memory hierarchy, CPU and CMP design spaces using Artificial Neural Networks (ANNs); Similarly, Lee et al. [23] proposed to use spline-based regression to predict performance and power from a large design space. It's also possible to derive optimal points based on their predictive models (e.g. via exhaustive prediction

in Pareto Analysis [24]), but our method is a one-step search that is more efficient and direct. Besides, PRIM can provide highly interpretable selective rules. More importantly, we demonstrated in this chapter that the PRIM-generated rules are effective across SPEC and SPLASH2 benchmarks. This is in contrast to traditional application-specific design space studies.

5.5 Conclusions

In this chapter, we propose to use a rule search statistical technique to generate simple selective rules on design parameters. These rules quantify the design space subregion that contains the configurations optimized for soft error reliability or other design metrics, providing computer architects with valuable guidelines to design reliable and high performance processors at early design stage. We found that reducing the AVF of a single processor structure may increase the vulnerability of other structures, and merely minimizing a processor's AVF may degrade performance. Our proposed generic approach is capable of generating a set of "universal" rules that achieves the optimization of the output variable across different programs in execution. The effectiveness of the universal rule set is validated on programs that are not used in training. Finally, the extension to multiprocessors enables a multi-objective optimization of reliability, performance and power for multi-threaded workloads.

CHAPTER 6. SUMMARY AND FUTURE WORK

6.1 Summary

Architectural Vulnerability Factor (AVF) characterizes a computer system's vulnerability to soft errors at architectural level. The scaling processor feature size, lower threshold voltage, and increasing clock frequency make current processors highly vulnerable to soft errors. Therefore, AVF provides useful guidelines to computer architects in designing reliable and high performance computer systems. However, measuring the AVF, either at an early design stage or during program runtime, incurs significant overhead in hardware structures and computation work. Consequently, this dissertation proposes a series of soft error resilient mechanisms via modeling and predicting the AVF using advanced statistical techniques.

First, we propose to use Boosted Regression Trees (BRT), a nonparametric tree-based predictive modeling scheme, to identify the correlation across workloads, execution phases, and processor configurations between a key processor structure's AVF and various performance metrics. The proposed method not only makes accurate prediction but also quantitatively illustrates individual performance variables' importance to the AVF. A quantitative comparison between our model and conventional linear regression is performed to demonstrate that our model shows higher stability when the model size varies. Moreover, to reduce the prediction complexity, we also utilize a rule search strategy named Patient Rule Induction Method (PRIM) to extract simple selective rules on important metrics, which can be dynamically applied during runtime to fast identify the vulnerable execution intervals. A case study that enables PRIM-based ROB redundancy is performed to demonstrate the applicability of the trained rules.

Second, the AVF measurement is even more complicated on multi-threaded processors such as Simultaneous Multithreading (SMT) and Chip Multiprocessor (CMP) architectures. The inter-thread resource contention shows significant and non-uniform impact on a program's AVF when it is co-scheduled with different programs. Furthermore, the data sharing in multi-threaded workloads may change the AVF behavior. Hence, we propose a scalable two-level predictive mechanism capable of predicting a program's AVF on a multi-threaded processor from easily measured metrics. Essentially, the first level model correlates the AVF in a contention-free environment with important performance metrics and the processor configuration; while the second level model captures the inter-thread resource contention and sharing via structure occupancies. By utilizing the proposed scheme, we can accurately estimate any unseen program's soft error vulnerability under resource sharing with any other program(s), on an arbitrarily configured multi-threaded processor. In practice, the proposed model can be used to find soft error resilient thread-to-core scheduling for a chip-multithreaded (CMT) processor.

Our third work in this dissertation improves the processor's soft error reliability at the pre-silicon stage. We propose to apply the PRIM method on processor design parameters. By exploring a large architectural design space, a set of selective rules on key design parameters can be generated to identify the configurations that are inherently reliable to soft errors. Our approach is capable of generating "universal" rules that achieve the optimization across different programs. The effectiveness of the universal rule set is further validated on programs not used in training. This cross-program capability is very useful in the era of multi-threading in terms of scalability and reducing training cost. Finally, the proposed scheme is extended to multiprocessors where multiple design metrics including reliability, performance and power are balanced. Our method-

ology is able to produce quantitative and universal solutions for both uniprocessors and multiprocessors.

6.2 Future Work

In the near future, I will continue to perform interdisciplinary research between computer architecture and experimental statistics, in which case finding a critical problem whose solution will be significantly improved with a more scientific and statistical approach is always important. In the long term, I am interested in building a reliable and high performance framework for future computer systems, including many-core processors, supercomputers, and GPUs.

6.2.1 Application Level Soft Error Propagation Analysis

The previous chapters examine the soft errors occurred in hardware at computer architecture level; on the other hand, we are also interested in the soft errors exposed to the application. Understanding how errors propagate through an application is important for reliable software design, especially if these errors are difficult to be detected and quantified. Existing works examining application level soft error vulnerability usually employ a fault injection approach: injecting random errors into the application and evaluating their effect on the output. However, this approach only shows application specific characteristics since the fault injections are only performed to this particular application. It becomes extremely expensive if a large number of applications need to be analyzed.

Therefore, we are developing a *generic* approach capable of examining and predicting soft error vulnerability for an *arbitrary* application. This approach is called *modular analysis*, isolating an application into individual routines and performing prediction for each of them. Specifically, an arbitrary application can be considered as a directed graph of routines: each of these

routines takes its input from another routine or the initial application input, and produces its own output that will be used as the input to another routine or the final output of the application. As illustrated in Figure 6-1, if an error occurs in one of these routines, it may propagate along with the routine dependency graph to the final application output.

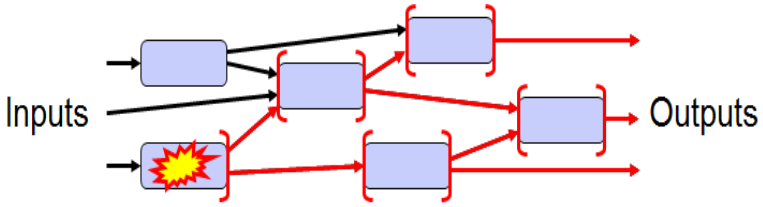

Figure 6-1. An application is a directed graph of routines

We propose to train a predictor from the input error pattern to the output error pattern for each routine. Any well-developed machine learning technique, such as Neural Networks or Support Vector Machine, can be used for predictor training. Consequently, for an arbitrary application, as long as the dependency among routines and the predictors for all involved routines are available, we can make an *end-to-end* prediction from the initial input error pattern to the final output error pattern of the application. This effectively gives the soft error vulnerability of the entire application.

Since a great number of High Performance Computing (HPC) programs make significant use of a small number of libraries, this modular approach is very efficient in analyzing HPC program vulnerabilities. Besides, it also provides compile-time guidelines to programmers about the soft error vulnerability of their programs, so that they can modify the program (e.g. via using a different algorithm) if necessary to improve the reliability.

6.2.2 Inter-Layer Coordinated Fault Tolerance

To evaluate a certain technique, one should look at its overall effect on the entire computer system, which spans multiple layers such as hardware, architecture, OS, and application layers. Traditional computer architecture research merely focuses on the architectural level, thus likely resulting in ultimate sub-optimal designs whose benefits from an innovative architectural scheme are diminished at upper layers. Therefore, we plan to investigate the interaction among different layers, developing fault tolerance techniques via inter-layer coordination.

One specific example is to generate reliable object code through compiler and architecture co-design. Current optimizing compilers usually have a large number of optimizations, each parameterized by a number of heuristics, thresholds, and flags. These parameters form a huge compiler design space, which can be combined with a processor configuration design space to derive the optimal solutions across both compiler and architecture layers. Furthermore, traditional operating systems employ optimization schemes that merely enhance performance; however, future systems will take other design metrics (e.g. reliability and power efficiency) into account. As a result, a variety of such schemes need to be revisited or redesigned to fulfill new requirements. Providing architectural support for this purpose will be one of the future studies.

6.2.3 Constructing Reliable High Performance Computing (HPC) Systems

A large scale HPC system consists of a great amount of processor cores, routers, disk arrays, etc. Failures of these small components super-linearly decrease the Mean-Time-To-Failure (MTTF) of the entire HPC system. Consequently, unexpected errors and exceptions are frequently encountered on supercomputers/clusters. Improving the reliability of HPC systems is necessary and urgent.

A fundamental challenge in conducting HPC studies is the lack of an architectural simulator capable of running HPC applications. I have been using a well-developed full-system simulator (i.e. M5 [2]) for my PhD research, and will extend this framework to simulate common HPC applications. With this tool, we will be able to develop architectural mechanisms that support parallel programming. For instance, we can examine and compare different implementations or algorithms of the same functional module used in an HPC application. Alternatively, we can also perform reliability studies at the application level. An example would be experiments of fault injections into commonly used routines of large scientific and commercial programs. We will characterize and predict the properties of error propagation through a series of routines in an HPC application.

REFERENCES

[1] T. Austin. DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design. In *International Symposium on Microarchitecture (MICRO)* 1999.

[2] N. Binkert et al. The M5 Simulator: Modeling Networked Systems. In *IEEE Micro*, vol. 26, no. 4, pp. 52-60, July – Aug. 2006.

[3] A. Biswas, P. Racunas, R. Cheveresan, J. Emer, S. Mukherjee, and R. Rangan. Computing Architectural Vulnerability Factors for Address-Based Structures. In *International Symposium on Computer Architecture (ISCA)* 2005.

[4] A. Biswas, P. Racunas, J. Emer, and S. Mukherjee. Computing Accurate AVFs using ACE Analysis on Performance Models: a Rebuttal. In *Computer Architecture Letters* Vol. 7, 2008.

[5] L. Breiman, J. Friedman, R. Olshen, and C. Stone. Classification and Regression Trees. Wadsworth International Group, Belmont, California, 1984.

[6] L. Breiman. Heuristics of Instability and Stabilization in Model Selection. In *Analysis of Statistics*, vol. 24, pp. 2350-2383, 1996.

[7] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting Inter-Thread Cache Contention on a Chip Multi-Processor Architecture. In *International Symposium on High-Performance Computer Architecture (HPCA)* 2005.

[8] C. Cho, W. Zhang, and T. Li. Informed Microarchitecture Design Space Exploration using Workload Dynamics. In *International Symposium on Microarchitecture (MICRO)* 2007.

[9] C. Cho, W. Zhang and T. Li. Modeling and Analyzing the Effect of Microarchitecture Design Space Parameters on Microprocessor Soft Error Vulnerability. In *MASCOTS* 2008.

[10] R. Deskan, D. Burger, S. Keckler, and T. Austin. Sim-alpha: A Validated, Execution-Driven Alpha 21264 Simulator. Tech Report TR-01-23, The University of Texas at Austin, 2001.

[11] L. Duan, B. Li, and L. Peng. Versatile Prediction and Fast Estimation of Architectural Vulnerability Factor from Processor Performance Metrics. In *International Symposium on High-Performance Computer Architecture (HPCA)* 2009.

[12] L. Duan, B. Li, and L. Peng. Reliability-Constrained Processor Performance Optimization via Design Parameter Selection. Poster in *PACT* 2009.

[13] L. Duan, L. Peng, and B. Li. Two-Level Soft Error Vulnerability Prediction on SMT/CMP Architectures. Short paper in *IEEE International Symposium on Workload Characterization (IISWC)*, 2011.

[14] L. Duan, Y. Zhang, B. Li, and L. Peng. Universal Rules Guided Design Parameter Selection for Soft Error Resilient Processors. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2011.

[15] B. Efron and R. Tibshirani. An Introduction to the Boostrap. Chapman & Hall/CRC, 1994.

[16] J. Friedman. Greedy Function Approximation: A Gradient Boosting Machine. In *The Annuals of Statistics*, 29, 1189-1232, 2001.

[17] J. Friedman and N. Fisher. Bump Hunting in High-dimensional Data. In *Statistics and Computing*, 9, 123-143, 1999.

[18] X. Fu, T. Li, and J. Fortes. Sim-SODA: A Unified Framework for Architectural Level Software Reliability Analysis. In *Workshop on Modeling, Benchmarking and Simulation (MoBS)* 2006.

[19] X. Fu, J. Poe, T. Li, and J. Fortes. Characterizing Microarchitecture Soft Error Vulnerability Phase Behavior. In *MASCOTS* 2006.

[20] M. Gomaa, C. Scarbrough, T. Vijaykumar, and I. Pomeranz. Transient-Fault Recovery for Chip Multiprocessors. In *ISCA* 2003.

[21] Y. Jiang, X. Shen, C. Jie, and R. Tripathi. Analysis and Approximation of Optimal Co-Scheduling on Chip Multiprocessor Models. In *PACT* 2008.

[22] E. Ipek, S. McKee, B. de Supinski, M. Schulz, and R. Caruana. Efficiently Exploring Architectural Design Spaces via Predictive Modeling. In *ASPLOS* 2006.

[23] B. Lee and D. Brooks. Accurate and Efficient Regression Modeling for Microarchitectural Performance and Power Prediction. In *ASPLOS* 2006.

[24] B. Lee and D. Brooks. Illustrative Design Space Studies with Microarchitectural Regression Models. In *HPCA* 2007.

[25] B. Lee and D. Brooks. Roughness of Microarchitectural Design Topologies and Its Implications for Optimization. In *HPCA* 2008.

[26] B. Lee, J. Collins, H. Wang, and D. Brooks. CPR: Composable Performance Regression for Scalable Multiprocessor Models. In *MICRO* 2008.

[27] B. Li, L. Duan, and L. Peng. Efficient Microarchitectural Vulnerabilities Prediction Using Boosted Regression Trees and Patient Rule Inductions. In *IEEE Transactions on Computers – Special Issue on System Level Design of Reliable Architectures*, vol 59-5, 2010.

[28] X. Li, S. Adve, P. Bose, and J. Rivers. Online Estimation of Architectural Vulnerability Factor for Soft Errors. In *ISCA* 2008.

[29] X. Li, S. Adve, P. Bose, and J. Rivers. SoftArch: An Architecture-Level Tool for Modeling and Analyzing Soft Errors. In *International Conference on Dependable Systems and Networks (DSN)* 2005.

[30] Y. Li, B. Lee, D. Brooks, Z. Hu and K. Skadron. CMP Design Space Exploration Subject to Physical Constraints. In *HPCA* 2006.

[31] M. Monchiero, R. Canal and A. Gonzalez. Design Space Exploration for Multicore Architectures: Power/Performance/Thermal View. In *ICS* 2006.

[32] S. Mukherjee, M. Kontz, and S. Reinhardt. Detailed Design and Evaluation of Redundant Multithreading Alternatives. In *ISCA* 2002.

[33] 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 *MICRO* 2003.

[34] J. Neter, et al. Applied Linear Statistical Models, fourth edition, McGraw-Hill/Irwin, Feb. 1996.

[35] S. Reinhardt and S. Mukherjee. Transient Fault Detection via Simultaneous Multithreading. In *ISCA* 2000.

[36] E. Rotenberg. AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessor. In *Fault-Tolerant Computing Systems (FTCS)* 1999.

[37] J. Sharkey, D. Ponomarev, and K. Ghose. M-SIM: A Flexible, Multithreaded Architectural Simulation Environment. Technical Report CS-TR-05-DP01, Department of Computer Science, SUNY at Binghamton, Oct. 2005.

[38] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically Characterizing Large Scale Program Behaviors. In *ASPLOS* 2002.

[39] SimpleScalar. www.simplescalar.com

[40] A. Snavely and D. Tullsen. Symbiotic Jobscheduling for a Simultaneous Multithreading Processor. In *ASPLOS* 2000.

[41] N. Soundararajan, A. Parashar, and A. Sivasubramaniam. Mechanisms for Bounding Vulnerabilities of Processor Structures. In *ISCA* 2007.

[42] N. Soundararajan, A. Sivasubramaniam, and V. Narayanan. Characterizing the Soft Error Vulnerability of Multicores Running Multithreaded Applications. In *SIGMETRICS* 2010.

[43] L. Spracklen and S. Abraham. Chip Multithreading: Opportunities and Challenges. In

HPCA 2005.

[44] V. Sridharan and D. Kaeli. Eliminating Microarchitectural Dependency from Architectural Vulnerability. In *HPCA* 2009.

[45] V. Sridharan and D. Kaeli. Using Hardware Vulnerability Factors to Enhance AVF Analysis. In *ISCA* 2010.

[46] T. Vijaykumar, I. Pomeranz, and K. Cheng. Transient-Fault Recovery Using Simultaneous Multithreading. In *ISCA* 2002.

[47] K. Walcott, G. Humphreys, and S. Gurumurthi. Dynamic Prediction of Architectural Vulnerability from Microarchitectural State. In *ISCA* 2007.

[48] N. Wang, A. Mahesri, and S. Patel. Examining ACE Analysis Reliability Estimates Using Fault-Injection. In *ISCA* 2007.

[49] N. Wang, J. Quek, T. Rafacz, and S. Patel. Characterizing the Effects of Transient Faults on a High-Performance Processor Pipeline. In *DSN* 2004.

[50] C. Weaver, J. Emer, S. Mukherjee, and S. Reinhardt. Techniques to Reduce the Soft Error Rate of a High-Performance Microprocessor. In *ISCA* 2004.

[51] S. C. Woo and et al. The SPLASH-2 Programs: Characterizing and Methodological Considerations. In *ISCA* 1995.

[52] W. Zhang, X. Fu, T. Li, and J. Fortes. An Analysis of Microarchitecture Vulnerability to Soft Errors on Simultaneous Multithreaded Architectures. In *ISPASS* 2007.

[53] W. Zhang and T. Li. Managing Multi-Core Soft-Error Reliability Through Utility-driven Cross Domain Optimization. In *ASAP* 2008.

[54] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing Shared Resource Contention in Multicore Processors via Scheduling. In *ASPLOS* 2010.

[55] J. Ziegler and et al. IBM Experiments in Soft Fails in Computer Electronics (1978-1994). IBM Journal of Research and Development, Volume 40, Number 1, 1996.

APPENDIX A. PERMISSIONS TO USE COPYRIGHTED MATERIALS

from j.hansson@ieee.org

reply-to Copyrights@ieee.org

to Lide Duan <lduan1@tigers.lsu.edu>

date Tue, Oct 25, 2011 at 10:16 AM

subject Re: Permission to reprint an IEEE TC article in dissertation

mailed-by ieee.org

Comments/Response to Case ID: 005E8DC9

ReplyTo: Copyrights@ieee.org

From: Jacqueline Hansson	**Date:** 10/25/2011
Subject: Re: Permission to reprint an IEEE TC article in dissertation	**Send To:** Lide Duan <lduan1@tigers.lsu.edu>
	cc:

Dear Lide Duan:

The IEEE does not require individuals working on a thesis to obtain a formal reuse license, however, you may print out this statement to be used as a permission grant:

Requirements to be followed when using any portion (e.g., figure, graph, table, or textual material) of an IEEE copyrighted paper in a thesis:

1) In the case of textual material (e.g., using short quotes or referring to the work within these papers) users must give full credit to the original source (author, paper, publication) followed by the IEEE copyright line © 2010 IEEE.

2) In the case of illustrations or tabular material, we require that the copyright line © [Year of original publication] IEEE appear prominently with each reprinted figure and/or table.

3) If a substantial portion of the original paper is to be used, and if you are not the senior author,

also obtain the senior author's approval.

Requirements to be followed when using an entire IEEE copyrighted paper in a thesis:

1) The following IEEE copyright/ credit notice should be placed prominently in the references: © 2010 IEEE. Reprinted, with permission, from [author names, paper title, IEEE publication title, and month/year of publication]

2) Only the accepted version of an IEEE copyrighted paper can be used when posting the paper or your thesis on-line.

3) In placing the thesis on the author's university website, please display the following message in a prominent place on the website: In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of Louisiana State University's products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to http://www.ieee.org/publications_standards/publications/rights/rights_link.html to learn how to obtain a License from RightsLink.

If applicable, University Microfilms and/or ProQuest Library may supply single copies of the dissertation.

For your future requests for IEEE copyrighted material, IEEE is pleased to announce our partnership with Copyright Clearance Center's RightsLink ®service. RightsLink offers a fast and easy way to obtain permission to reuse and republish material from IEEE. You can go to RightsLink for your requests for IEEE copyrighted material, by following these simple instructions:

1. Please go to http://ieeexplore.ieee.org/Xplore/guesthome.jsp
2. Type the title of the first IEEE copyrighted paper into the Search Bar.
3. The IEEE copyrighted paper's title will appear in the Search Results. Click on the underlined title.
5. Next, click on the orange "Request Permissions" link, which can be found under the IEEE paper title.
6. Create a RightsLink account, select the reuse preferences and obtain permission for the IEEE content requested.
7. Continue the process for any material desired from each IEEE copyrighted paper.

If you have any questions or comments about this process, please direct them to pubs-permissions@ieee.org. Thank you.

Sincerely,

Jacqueline Hansson, Coordinator

©©©©©©©©©©©©©©©©©©©©©©©©©©©©©©©©
IEEE Intellectual Property Rights Office
445 Hoes Lane
Piscataway, NJ 08855-1331 USA
+1 732 562 3966 (phone)
+1 732 562 1746(fax)

IEEE Fostering technological innovation
and excellence for the benefit of humanity.

©©©©©©©©©©©©©©©©©©©©©©©©©©©©©©©©©

Dear Sir/Madam,

My name is Lide Duan, and I am completing my PhD dissertation titled "Analyzing and Predicting Processor Vulnerability to Soft Errors Using Statistical Techniques" at Louisiana State University. I am writing to obtain your permission to reprint one of my papers appeared in IEEE Transactions on Computers as one chapter in my dissertation. The detailed information about this paper is the following:

B. Li, L. Duan, and L. Peng, "Efficient Microarchitectural Vulnerabilities Prediction Using Boosted Regression Trees and Patient Rule Inductions," In IEEE Transactions on Computers (TC) – Special Issue on System Level Design of Reliable Architectures, vol. 59(5), pp. 593-607, May 2010.

Please let me know if you have any questions. Your prompt response would be very appreciated! Thanks!

Regards,
Lide

APPENDIX B. AUTHOR'S PUBLICATIONS

- (Under Review) **L. Duan**, L. Peng, and B. Li, "Predicting Architectural Vulnerability on Multi-Threaded Processors with Resource Sharing," Submitted for review, 2011.
- **L. Duan**, L. Peng, and B. Li, "Two-Level Soft Error Vulnerability Prediction on SMT/CMP Architectures," In Proceedings (short paper) of *IEEE International Symposium on Workload Characterization (IISWC)*, Austin, TX, Nov. 2011.
- **L. Duan**, Y. Zhang, B. Li, and L. Peng, "Universal Rules Guided Design Parameter Selection for Soft Error Resilient Processors," In Proceedings of *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Austin, TX, Apr. 2011.
- B. Li, **L. Duan**, and L. Peng, "Efficient Microarchitectural Vulnerabilities Prediction Using Boosted Regression Trees and Patient Rule Inductions," In *IEEE Transactions on Computers (TC) – Special Issue on System Level Design of Reliable Architectures*, vol. 59(5), pp. 593-607, May 2010.
- Y. Zhang, L. Peng, W. Lu, **L. Duan**, and S. Rai, "Expediating IP Lookups with Reduced Power via TBM and SST Supernode Caching," In *Computer Communications*, vol. 33(3), pp. 390-397, Feb. 2010.
- **L. Duan**, B. Li, and L. Peng, "Reliability-Constrained Processor Performance Optimization via Design Parameter Selection," In Poster session of *The 18th IEEE/ACM International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Raleigh, NC, Sep. 2009.
- **L. Duan**, B. Li, and L. Peng, "Versatile Prediction and Fast Estimation of Architectural Vulnerability Factor from Processor Performance Metrics," In Proceedings of *The 15th*

IEEE International Symposium on High-Performance Computer Architecture (HPCA), Raleigh, NC, Feb. 2009.

- L. Peng, W. Lu, and **L. Duan**, "Power Efficient IP Lookup with Supernode Caching," In Proceedings of *The 50th IEEE Global Communications Conference (Globecom),* Washington D.C., Nov. 2007.

VITA

Lide Duan was born on September 23rd, 1983, in Longyan, Fujian, China. He received his Bachelor of Science degree in Computer Science and Engineering from Shanghai Jiao Tong University, Shanghai, China, in June 2006. Since then, he has been enrolled in the Department of Electrical and Computer Engineering at Louisiana State University, Baton Rouge, Louisiana, to pursue his doctorate degree. During this period, he passed his qualify exam in Fall 2008 and general exam in December 2009, respectively. He will graduate with his Doctor of Philosophy degree in Fall 2011.

Lide's research interests broadly lie in the area of computer architecture with particular emphasis on processor reliability against soft errors. He has published a series of papers on this topic in various computer architecture conferences and journals. He also finished a research internship at Lawrence Livermore National Laboratory, Livermore, California, from May 2011 to August 2011, working on application level error propagation analysis and prediction.