ABSTRACT

Title of dissertation: Exploiting Inherent Program Redundancy for Fault Tolerance

Xuanhua Li, Doctor of Philosophy, 2009

Dissertation directed by: Professor Donald Yeung
Department of Electrical and Computer Engineering

Technology scaling has led to growing concerns about reliability in microprocessors. Currently, fault tolerance studies rely on creating explicitly redundant execution for fault detection or recovery, which usually involves expensive cost on performance, power, or hardware, etc. In our study, we find exploiting program's inherent redundancy can better trade off between reliability, performance, and hardware cost.

This work proposes two approaches to enhance program reliability. The first approach investigates the additional fault resilience at the application level. We explore program correctness definition that views correctness from the application's standpoint rather than the architecture's standpoint. Under application-level correctness, multiple numerical outputs can be deemed as correct as long as they are acceptable to users. Thus faults that cause program to produce such outputs can also be tolerated. We find programs which produce inexact and/or approximate outputs can be very resilient at the application level. We call such programs *soft computations*, and find that they are common in multimedia workloads, as well as

artificial intelligence (AI) workloads. Programs that only compute exact numerical outputs offer less error resilience at the application level. However, *all* programs that we have studied exhibit some enhanced fault resilience at the application level, including those that are traditionally considered as exact computations–*e.g.*, SPECInt CPU2000.

We conduct fault injection experiments and evaluate the additional fault tolerance at the application level compared to the traditional architectural level. We also exploit the relaxed requirements for numerical integrity of application-level correctness to reduce checkpoint cost: our lightweight recovery mechanism checkpoints a minimal set of program state including program counter, architectural register file, and stack; our soft-checkpointing technique identifies computations that are resilient to errors and excludes their output state from checkpoint. Both techniques incur much smaller runtime overhead than traditional checkpointing, but can successfully recover either all or a major part of program crashes in soft computations.

The second approach we take studies value predictability for reducing fault rate. Value prediction is considered as additional execution, and its results are compared with corresponding computational outputs. Any mismatch between them is accounted as symptom of potential faults and incurs restoration process. To reduce misprediction rate caused by limitations of predictor itself, we characterize fault vulnerability at the instruction level and only apply value prediction to instructions that are highly susceptible to faults. We also vary threshold of confidence estimation according to instruction's vulnerability–instructions with high vulnerability are assigned with low confidence threshold, while instructions with low vulnerability

are assigned with high confidence threshold. Our experimental results show benefit from such selective prediction and adaptive confidence threshold on balance between reliability and performance.

Exploiting Inherent Program Redundancy for Fault Tolerance

by

Xuanhua Li

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2009

Advisory Committee:
Professor Donald Yeung, Chair/Advisor
Professor Gang Qu
Professor Manoj Franklin
Professor Bruce Jacob
Professor Amr Baz

Dedication

For my lovely baby Kelly.

Acknowledgments

I sincerely express my gratitude to all the people who have made this thesis possible. Because of their kindness and love, my graduate experience has been one that I will cherish forever.

First of all, I really thank my advisor, Dr. Donald Yeung. He has not only patiently advised my research throughout my long stay in the graduate program, but also impressed me for life with his own integrity, intelligence, passion to work, etc. I always enjoyed the time discussing problems with him. And his help and advices have helped me to overcome numerous difficulties.

I also thank my dear husband, Huanfeng, and my family in China. I always thank God for bringing Huanfeng to my life. He is such a wonderful husband, a good friend and listener. I know no matter what happens, he is always there supporting me. I also thank my own family–my dad and mom, my brother and sister-in-law, and my little niece. Although they are far away from me, their consistent love and caring have supported me throughout the past years. I especially thank my brother, Menghua, for all he has done and is doing for our family.

I always feel lucky to be one of the members in Systems and Computer Architecture Lab (SCAL). I thank my colleagues for their help and discussion: Seungryul, Dongkeun, Hameed, Wanli, Meng-ju, Inseok, Xu, Aamer, Brinda, Rania, Kursad, Sumesh, Ohm, Zahran, and all other lab-mates. I also appreciate the members of my PhD defense committee: Dr. Gang Qu, Dr. Manoj Franklin, Dr. Bruce Jacob, and Dr. Amr Baz. I would like to acknowledge other graduate students and staffs at

Maryland: Bo, Juanjuan, and Vivian. I also express my special thanks to friends I knew during my internships: Dr. Steve Crago, Janice McMahon, Paul, Saila, Hima, and many others. I apologize to those I have inadvertently left out.

In addition, I thank my baby who is coming to this world soon. I am so excited and looking forward to embracing her.

Lastly, I sincerely thank God for his endless love and care.

Table of Contents

List of Figures

List of Abbreviations

AI	Artificial Intelligence
IPC	Instructions per Cycle
SEU	Single-Event Upset
MTTF	Mean Time to Failure
MTBF	Mean Time between Failures
MITF	Mean Instructions to Failure
AVF	Architectural Vulnerability Factor

Chapter 1

Introduction

1.1 Motivation

As CMOS technology scaling continues to enable faster transistors and lower supply voltage, tremendous improvements have been brought to microprocessor performance and power consumption. However, one of the most striking downsides of these trends is that computer systems become significantly more susceptible to hardware faults–particularly, soft errors (also known as transient faults). As intermittent faults, soft errors arise from strikes by cosmic particles and radiation from packaging materials, and are more seriously endangering system reliability as the number of on-chip transistors keeps growing exponentially. It is estimated that a chip's error rate will scale in proportion to the number of devices–*i.e.*, with Moore's Law [2].

To detect or recover from faults, there are basically two kinds of approaches. The traditional approach explicitly replicates program computation or program state, and uses the replicated program copy to detect or recover fault corruptions. For example, to detect potential faults, additional hardware structures such as processor cores, hardware contexts, and functional units [3, 4, 5, 6, 7] can be utilized, or alternatively, program code is duplicated during compiling stage [8, 9, 10], thus program computational outputs can be compared and checked for corruption. In addition, for recovering potential faults, checkpointing is usually adopted to create

additional copies of program state. Such duplication of program state can then be used for rolling back program execution once a fault is detected. Unfortunately, for both fault detection and recovery, although such approach of explicit replication can be very accurate, it usually incurs a significant cost on either performance, power, or hardware expense, etc.

Contrastingly, a relatively new approach exploits programs' inherent redundancy to improve fault tolerance, such as the technique that use symptoms like branch misprediction as indication of possible fault corruptions [11]. Another technique, which is called fault screening, identifies value space of instructions' computation, and triggers recovery if the instructions' future outputs are not within the recorded space [12]. Compared to the traditional approach, these techniques do not replicate program computation or state, hence they incur very low overhead. But they are not perfect–either the fault checkers are too sensitive and catch false positives, or they can not recover all possible faults. We call such techniques *probabilistic fault tolerance*. Although the probabilistic techniques cannot achieve failure-safe execution, they are still very useful for most general-purpose systems which do not need perfect fault coverage but are very sensitive to cost. For such systems, these techniques can reduce fault rate with small cost.

In this thesis, we contribute to existing probabilistic techniques by exploiting two new sources of inherent program redundancy to improve fault tolerance [13, 14]. First, we find that many programs compute much more precisely than necessary. Although traditional approaches to evaluating correctness require numerical integrity of architectural state, in many cases, it is not necessary for program state to be

numerically correct. Instead, program data may tolerate some errors, or there may exist multiple values which all lead to correct program execution from the user's standpoint. This program characteristic is highly application dependent but we find it occurs frequently in application domains that involve sensing data, such as multimedia and artificial intelligence (AI) workloads. In such workloads, as long as programs' final outputs meet user's requirement, they are acceptable even if their numerical values are different from their fault-free versions. For example, some precision loss in outputs is tolerable to users if their precision requirements are lower than the datatypes supported by the programming environment or hardware architecture. Also, program solutions that are not optimal but still adequate from the user's standpoint can be accepted as well. In our study, we call such standard of correctness which is evaluated from the user or application level as *application-level correctness*. In contrast to the traditional definition that views correctness at the architecture level, application-level correctness is examined at a higher level of abstraction, thus allowing the existence of multiple correct outputs. Faults that cause program execution to be numerically incorrect, but still produce one of the outputs that are acceptable to users, can be tolerated.

In addition to the data redundancy at the application level, program computations can also show a high degree of redundancy–*i.e.*, instruction streams and their output results often exhibit repeatability. Such redundancy has been traditionally exploited for increasing instruction-level parallelism (ILP) to improve program performance. One of the best examples of this is value prediction which has been widely studied for boosting program performance, but has not yet been applied to

fault tolerance. In our study, we find value prediction can be used to detect faults–prediction results can be compared against the actual computation results, with comparison mismatches indicating potential data corruptions.

In our study, we find it is beneficial to exploit both sources of program's inherent redundancy for fault tolerance enhancement. In particular, we exploit redundancy at the application level for lightweight fault-recovery mechanisms–our selective checkpointing technique only checkpoints a small part of program state while it can successfully recover a large portion or most of program crashes. We also exploit value predictability for low-cost fault detection–we take value prediction as another form of program execution, and compare the results of value prediction and actual computation to detect potential faults. In addition, we implement experiments to explore the tradeoff between performance and reliability impacts.

The following texts briefly introduce application-level fault tolerance (Section 1.1.1) and value prediction (Section 1.1.2) that have initiated our work.

1.1.1 Application-level Fault Tolerance

First, as we have discussed, application-level correctness enables the numerical redundancy of program state from user's standpoint. Compared to the traditional architecture-level definition of correctness, application-level correctness allows more faults to be tolerable as long as the final outputs can be accepted by users. Although such property of additional fault tolerance at the user level can be found in many important workloads, the degree of tolerating faults is application-specific.

Different programs have different characteristics, thus they may appear more or less resilient to faults. For instance, multimedia programs process human sensory and perception information and are highly fault resilient at the application level. Another example is artificial intelligence (AI) workloads, which have shown more significance recently [15]. AI is a vast research area and includes many branches such as reasoning, inference, and machine learning. As we will discuss later, AI algorithms also exhibit a great deal of fault tolerance at the application level. All these programs, *e.g.*, multimedia and AI programs, belong to a class of computations which we call *soft computations* [16, 17]. Soft computations compute on inexact or approximate data. Their outputs are associated with certain forms of qualitative representations, which are usually interpreted by users. Certain faults which may change the numerical values of those outputs do not change the corresponding qualitative answers, thus are tolerable to users. Compared to soft computations, programs which requires numerically exact outputs in order to be correct, such as traditional scientific computations, may offer much less amount of fault resilience at the application level.

Although the degree of error resilience at the application level varies across different applications, we find *all* programs that we have studied exhibit some enhanced fault tolerance from user's standpoint, including those that are traditionally considered as exact computations–*e.g.*, SPECInt CPU2000. Such application-level fault tolerance could be exploited to avoid overdesign and achieve better tradeoff between system reliability and performance cost.

1.1.2 Value Prediction-based Fault Detection

Prior studies have shown that program execution including its instruction and data streams exhibits another kind of redundancy–repeatability. One example of applying such inherent redundancy is value prediction, which predicts instruction results by learning their past values, thus breaks true data dependency. Unfortunately, although value prediction seems promising, its effectiveness is weakened due to its high misprediction penalty, which turns to be even worse as processor pipeline becomes deeper.

In our study, we find such predictability in computational outputs can be exploited to improve system reliability. Compared to other redundant-execution techniques, the biggest advantage of applying value prediction for result comparison and fault detection is, by exploiting program's inherent redundancy, it avoids the need for explicitly duplicating hardware or program execution, thus evades related area and power demands or performance degradation. Although value predictor itself requires some additional hardware, we find a relatively small predictor is effective in detecting faults. In addition, compared to the traditional applications of value prediction which mainly pursue performance speedup, in the field of fault tolerance, it is acceptable to trade between performance and reliability impact, which brings more chances for value prediction. For example, in case of value misprediction, we adopt the idea of flushing pipeline to try to recover fault corruptions. Although flushing degrades program performance, it improves reliability when fewer valid bits become vulnerable during flushing and re-execution. Moreover, unlike traditional

applications that require value prediction results at the early stage of pipeline, for detecting faults, value prediction can wait until the end of execution stage. All these render much flexibility for adapting value predictor design to improve fault tolerance.

In addition, by characterizing fault vulnerability at the instruction level–*i.e.*, sorting instructions by how much they contribute to a hardware structure's fault vulnerability during program execution, we find for the hardware structures we have studied, a small portion of instructions accounts for a major fraction of program vulnerability. For example, for fetch buffer in our processor model, about 3.5% instructions contribute to 53.9% of total AVF for SPEC2000 benchmark TWOLF. Therefore, by selectively protecting such a small portion of instructions from fault corruption, the overall reliability can be enhanced greatly, while program performance is much less affected than protecting all the instructions. We apply such idea into our experiments–selectively predicting the most vulnerable instructions–to achieve better fault coverage with smaller performance impact.

To further reduce performance cost caused by value misprediction and the consequent pipeline flush, we utilize confidence estimator which allows value prediction only when there is enough confidence in its correctness. We also incorporate the fact that instructions do not contribute equally to reliability, and vary confidence threshold accordingly–instructions that are more susceptible to faults use lower confidence threshold, and instructions that are less susceptible use higher confidence threshold.

1.2 Contributions

This dissertation makes the following contributions.

Complete Study on Application-Level Correctness. In this work, we present our study on exploring definitions of program correctness and their impacts on fault tolerance. To characterize program susceptibility under different correctness definitions, we implement fault injection experiments and measure how many more faults program can tolerate under application-level correctness, compared to the traditional architectural correctness. Our results show that for soft computations, about 45.8% of fault injections that lead to architecturally incorrect execution produce acceptable results under application-level correctness. For SPEC programs, a smaller portion of architecturally incorrect faults, 17.6%, produce acceptable results at the application level. Such results indicate the degree of program-level redundancy that provides additional fault tolerance.

Analysis on Sources of Application-Level Fault Tolerance. As discussed by Mukherjee *et al* [18] and Wang *et al* [19], there are sources of masking at the microarchitecture level as well as architecture level which reduce the probability that faults affect program's output. Our study exposes another level of fault masking– the application level–from user's standpoint. By analyzing program characteristics as well as how faults propagate, we identify several sources of redundancy at the application level which render additional fault tolerance.

New Fault Recovery Techniques by Exploiting Application-Level Correctness. We implement new fault recovery techniques by exploiting such additional redundancy at the application level. One technique we propose is stack checkpointing which only saves program counter, architectural register file, and stack. Our results show about 66.3% of program crashes in our multimedia and AI workloads can be successfully recovered. Another mechanism we propose identifies program state that are resilient to errors, which we call "soft" state. In our experiments, we incrementally checkpoint state that are not marked as soft state and have been updated during each checkpoint period. Our results show small additional performance cost, compared to lightweight fault recovery, while almost all crashes in soft computations are successfully recovered.

Characterizing Fault Vulnerability at the Instruction Level. In our study, we characterize instruction's vulnerability by computing the percentage of a hardware structure's average AVF (Architectural Vulnerability Factor) that an instruction relates to. We find that a small portion of instructions accounts for a major fraction of program vulnerability. Thus, by selectively protecting such small portion of instructions from fault corruption, the overall reliability can be enhanced greatly, while program performance is affected much less compared to fully protection.

New Fault Detection Technique by Exploiting Value Predictability. We apply value prediction for checking program computational results and consider misprediction as symptom of potential fault. In our work, we only predict instructions that have high vulnerability and squash pipeline on mispredictions to recover po-

tential faults. To reduce possible fault positives that are caused by limitation of value predictor itself, we implement confidence estimator to reduce misprediction rate. In addition, by incorporating various vulnerability at the instruction level, we propose adaptive confidence estimation–adjusting confidence threshold according to instruction's vulnerability. We analyze the impacts of value prediction and confidence estimation on both aspects of program reliability and performance.

1.3 Road Map

The rest of the dissertation is organized as follows.

Chapter 2 explains the background of our study about hardware faults and fault tolerance research.

Chapter 3 analyzes application-level correctness and characterizes fault susceptibility at the application level, compared to the traditional architectural level.

In Chapter 4, we present our lightweight fault recovery as well as soft-checkpointing recovery techniques.

Chapter 5 discusses our fault detection technique with value prediction. We apply our selective value prediction (by instruction vulnerability) and adaptive confidence mechanisms to detect potential faults. We evaluate both reliability and performance impacts of our technique. In addition, we also implement fault injection experiments to compare with other related work.

Chapter 6 lists the prior work related to our study, covering soft computation, fault susceptibility characterization, analysis of fault-tolerance sources, fault

detection and recovery techniques.

Finally, Chapter 7 concludes the dissertation and suggests the future research directions.

Chapter 2

Background

2.1 Hardware Faults

Hardware faults such as soft errors, lifetime reliability and process variation problems, have always posed threats to normal functionality of semiconductor-based digital systems. Such problems are becoming more serious as scaling in device size, operating voltages and design margins continues. The following describes the nature of these types of faults.

Transient Fault A transient fault, or soft error, is a signal or datum that is wrong, but is not caused by hardware defect–mistake in design or construction, or broken component. It is not a permanent failure–the occurrence of soft errors does not cause permanent damage to hardware, and there is no implication on reliability reduction of the system itself. On the contrary, it is temporary and intermittent fault, but can still corrupt normal program execution.

Soft errors are mainly due to particle strikes such as alpha particles emitted by decaying radioactive impurities in packaging and interconnect materials, energetic neutrons and protons from cosmic rays, or thermal neutrons. When particles travel through a semiconductor device, they release electron-hole pairs which can be absorbed by transistor source and drain nodes and disturb the distribution of

electrons there. If the disturbance is large enough, the state of a logic device–such as an SRAM cell, a latch or a gate–can be changed from a 0 to a 1 or vice versa.

Soft errors have become a more challenging problem for future microprocessor design. While the raw error rate per logic device remains small, a processor's error rate grows proportional to the number of transistors per chip which has kept scaling with the rapid technology development [2].

Usually, soft errors are localized to a very small area of a chip, and only affect the state of one logic node. This is known as a Single-Even Upset (SEU). We mainly study this form of fault in our work.

Lifetime Reliability System lifetime reliability is impacted by wear-out based failures, which are mainly caused by migration of metal atoms due to electro- or mechanical stress, gate oxide wear down, or damage accumulated by thermal cycling [20, 21].

Device scaling results in increased power density, and consequently, temperature, which directly affect processor lifetime reliability. For example, the main failure mechanisms including atom migration, gate oxide wear down and thermal cycling, are adversely affected by increases in temperature, while the decreasing feature size of interconnects accelerates failure rate due to electro-migration.

Process Variation Process variation is mainly caused by fluctuations in dopant concentrations and device channel dimensions, which cause deviations in the manufactured properties of the chip such as transistor size, threshold voltage or driving capability, etc., thus affecting stability of circuit blocks [22].

As devices scale with decreasing dimensions and growing transistor density, device variation as well as the probability of deviations in longer critical path delay increase, which will significantly affect performance, and more severely, compromises reliability.

2.2 Fault Tolerance

Current fault tolerance research focuses primarily on characterizing fault susceptibility or developing fault detection and recovery techniques. This section discusses the nature of research in these areas.

2.2.1 Fault Susceptibility Characterization

To characterize a device's susceptibility to faults, it is necessary to understand how faults propagate in circuits after they hit devices. Traditionally, soft errors were tackled within the context of memory cells, for which error detection and correction circuits are widely used for protection. Combinational logic circuits, on the other hand, have been found to be less susceptible to SEU due to the naturally occurring masking effects: electrical masking, logical masking, and temporal (or timing-window) masking. In particular, electrical masking attenuates affected signal by the electrical properties of gates on the propagation path such that the resulting pulse is of insufficient magnitude to be reliably latched; logical masking occurs when the propagation of an SEU is blocked from reaching an output latch because off-path gate inputs prevent a logical transition of that gate's output; and temporal masking

occurs when the erroneous pulse reaches an output latch, but it does not occur close enough to when the latch actually samples its input.

Although these masking effects reduce the probability of soft errors to be manifested, they are diminishing as device feature size decreases and circuits adopt higher operating frequencies. As a result, recently more and more work has been devoted to study the characteristics of fault susceptibility by building proper models or implementing fault injection experiments, etc.

In addition to masking effects at the circuit level, there exist other sources of masking such as those proposed by Mukherjee *et al* [18]. Mukherjee *et al* identified the effects of microarchitecture-level masking which come from mispeculated instructions, predictor structure bits, and microarchitecturally idle bits. They also identified masking effects at the architectural level–for example, faults on NOP instructions, performance-enhancing instructions, dynamically dead code, and logically masked instructions. Faults on such microarchitecture structure and architectural instructions do not affect program outputs.

To calculate fault susceptibility, there exist various metrics to specify Soft error rate (SER). SER is the rate at which a device or system encounters or is predicted to encounter soft errors. It is typically expressed as either the number of failures-in-time (FIT), or mean-time-between-failures (MTBF). The unit adopted for quantifying failures in time is called FIT, equivalent to 1 error per billion hours of device operation. MTBF is usually given in years of device operation. To put it in perspective, 1 year MTBF is equal to approximately 114,077 FIT.

MTBF can be further expressed as mean-time-to-failure (MTTF) and mean-

time-to-repair (MTTR). Generally MTTR is ignorable compared to MTBF. Also considering the fact that usually processor vendors have no control over factors related to MTTR, MTTF is more frequently used for featuring fault susceptibility.

2.2.2 Fault Detection

Fault Detection is the process of discovering if a fault has occurred. Several schemes exist to achieve fault detection. For example, error detection codes are often used in data storage media, in which extra bits, referred to as check bits, are utilized for storing information that is derived from data to be protected. Fault can be detected by re-generating the check bits and comparing with the old check bits–a mismatch on comparison indicates the occurrence of a fault. Examples of check bits include parity bits, checksum, etc.

For more comprehensive comparison, dual modular redundancy (DMR) can be employed at various levels to enable fault detection. DMR has duplicated elements which work in parallel–the duplicated elements can range from replicated pipelines within the same die to separate processors. At any time, all the replications of each element should be in the same state: the same inputs are provided to each element, and the same outputs are expected. To detect faults, the outputs of the replications are compared using a voting circuit. A fault is detected when mismatch on output comparison is captured. However, other methods are needed in addition to DMR for recovery.

2.2.3 Fault Recovery

Fault recovery is the process of limiting fault propagation and enabling the service which restores system to an acceptable state. It can be accomplished in two general ways. The first mechanism is forward-error recovery, in which enough redundancy exists in the system to determine the correct operation. For example, triple modular redundancy (TMR) utilizes three systems which perform the same functions. Results of the three composing units are processed by a voting system to produce a single output. Usually the voter is much more reliable than other TMR components. When any one of the three systems fails–its result is different from those produced by the other two systems. The remaining systems can correct and mask the fault.

Another recovery mechanism is backwards-error recovery, which creates checkpoints of system state and rollback program execution when an error is detected. Checkpointing can be implemented by operating system, or at the user level, incorporating special checkpointing mechanism with the application program. It is performed at a periodic interval by storing checkpoints to disk. If a failure occurs which causes the application to be terminated prematurely, the stored (most-recent) checkpoint can be used to restart the application–with the loss of computation during a checkpoint interval.

To checkpoint an application, its state including values in memory, CPU registers, and the state of the operating system such as the file system, have to be saved. Typically, the memory state can be divided into four parts: executable code, global

variables, heap and stack. The global variables, heap and stack need to be stored in checkpoint. As for the executable code, because it is usually unchanged since the beginning of execution, it can be restored from the program's executable file and thus does not need to be saved every time.

To reduce checkpoint size as well as runtime overhead, incremental checkpointing uses page protection hardware to identify and only save the portion of pages that have been updated since the previous checkpoint. Our recovery mechanism, which is to be described in Section 4, is based on incremental checkpointing.

2.2.4 Partial Fault Detection/Recovery

Traditional fault-tolerance techniques aim at achieving perfect coverage–detecting and recovering from all possible latent faults. However, for most systems such as desktop computers or commodity servers, such high reliability is not necessary. On the contrary, for those systems, performance and hardware cost appear to be more crucial for customers. Thus recently, a lot of work has been committed to reducing fault rate with low performance or hardware cost.

Chapter 3

Application-level Fault Susceptibility

In this chapter, we present our work on characterizing programs' fault susceptibility at the application level. We first discuss definitions of application-level correctness (Section 3.1). Then we report our experimental methodology as well as susceptibility results (Section 3.2). Lastly, we discuss various sources of fault tolerance at the application level (Section 3.3).

3.1 Application-Level Correctness

This section presents our study on application-level correctness. We first discuss qualitative program outputs (Section 3.1.1). Then, we present various correctness definitions which are used in our fault susceptibility experiments (Section 3.1.2).

3.1.1 Qualitative Program Outputs

Traditionally, program outputs are defined as all the numerical values that are stored to memory at program completion. And program's execution is said to be correct as long as all the output values are the same as those obtained by a fault-free execution. However, in many cases, only a small portion of the final values need to be provided to users. Thus, state that are invisible to users do not need to be correct. More interestingly, in a lot of cases, even the results that are

presented to users do not need to be exact either. This commonly occurs in programs computing results that are qualitatively interpreted by the user, such as multimedia workloads which process human sensory information. In these programs, different numerical results can lead to the same qualitative interpretation. Another example is AI processing that applies artificial intelligence algorithms for reasoning, inference, and learning, etc. These algorithms are approximate in nature, and errors in the numerical answers may not affect the programs on drawing further conclusions(*i.e.*, qualitative answers). For example, results of learning algorithms may be used for classifying new datasets. Thus, numerically different results from learning could be qualitatively correct if they lead to the same classification answers.

Not only soft computations like multimedia and AI programs have qualitative outputs, but other general computations may also exhibit such properties. For instance, many programs are designed to achieve the best performance, such as a compression algorithm that tries to generate an output file as compact as possible. However, the ultimate goal is to faithfully convert the compressed file back to the original data. Hence, even if the compression process is not efficient and produces a bigger compressed output, the user may still accept it if it leads to the same correct result after decompression.

Furthermore, even qualitative answers can vary if they are acceptable at the user(*i.e.*, application) level. Considering multimedia applications again, outputs which result in similar, although not the same, qualitative interpretation can be deemed as correct as long as users are satisfied. The amount of acceptable error in the output depends on users. This allows correctness to cease to be simply black or

white; instead, we can speak of a degree of correctness determined by the amount of tolerable error.

3.1.2 Correctness Definitions

Existing definitions of program correctness can be expanded by considering solution quality interpreted at the user level. Below are possible correctness definitions, listed in decreasing strictness.

I. Architectural state is numerically correct on a per-cycle (or per multiple-cycle) basis.

II. Output state (*i.e.*, architectural state visible after program completion) is numerically correct.

III. Output state is qualitatively correct with high fidelity.

IV. Output state is qualitatively correct with acceptable fidelity.

Definitions I and II are two main traditional approaches used in fault tolerance research. They require program state, either intermediate or final results, to be numerically exact compared to fault-free execution. Unlike the first two architecture-level definitions, the remaining ones, definitions III and IV, are at the application level, thus they are less strict. Both correctness definitions apply to programs producing results that have a higher qualitative interpretation. More specifically, definition III requires outputs to be the same or very similar to the baseline output(*i.e.*, high fidelity). The baseline output is defined as the result obtained from fault-free

execution of a program. In other words, answers satisfying definition III do not have noticeable quality degradation. Definition IV requires answers to have relatively good fidelity compared to baseline solutions. Compared to definition III, definition IV allows more tolerance of reduced answer quality.

As we have discussed in Section 3.1.1, programs with qualitative answers exhibit more error resilience since their results are not directly tied to precise numerical values. Errors in the numerical results may not be noticeable, or they may not significantly impact qualitative answers. Furthermore, under application-level correctness definitions, *i.e.*, definitions III and IV, the output acceptability can be customized by users by varying the minimum fidelity necessary for a program output to be "acceptable." Hence, users can accept more program outputs as "correct" by sacrificing fidelity. Generally, the more tolerable the qualitative answer is to error, the more reliable the program appears to be. As we will see in Section 3.2, this provides users with a great opportunity to tradeoff solution quality for fault tolerance.

3.2 Fault Susceptibility Experiments

This section discusses the impact of application-level correctness on fault susceptibility. We conduct experiments to quantify how much more fault resilient programs appear to be under application-level correctness compared to traditional architecture-level correctness. We first present our experimental methodology in Section 3.2.1, then discuss our results in Section 3.2.2.

3.2.1 Experimental Methodology

To analyze program fault susceptibility under different correctness definitions which have been described in Section 3.1.2, we perform statistical fault injection experiments on selected benchmarks and analyze the effects of faults on a microarchitecture model. We also assume a Single Event Upset, or SEU, fault model, throughout our experiments. In each fault injection experiment, a random single bit flip fault is injected into the execution of one of the benchmarks. In order to capture architectural effects of the faults more efficiently, we adopt the methodology introduced by Reis *et. al.* [23] which uses a two-phase simulation technique. Similar to [23], during the first fault injection phase, we inject faults into a detailed architectural simulator. Each time one fault is injected, we continue simulation until the effects of the fault have been completely manifested in architectural state. Then, we checkpoint the architectural effects, and continue to inject another fault. After we have compiled the information across all the fault injections, in the second phase, we use a simple functional simulator and replay each fault injection experiment that has been recorded. More specifically, for each fault, we resume simulation with its corresponding checkpoint, thus the architectural effects of the fault will keep propagating while the program executes. In each simulation trial, we try to run the program to completion. If the program does not crash, we evaluate its outputs under both architecture- and application-level correctness. Note in our experiments, we only evaluate program correctness with definitions II, III and IV introduced in Section 3.1.2.

Processor Parameters	
Bandwidth	8-Fetch, 8-Issue, 8-Commit
Queue size	64-IFQ, 40-Int IQ, 30-FP IQ, 128-LSQ
Rename register / ROB	128-Int, 128-FP / 256 entry
Functional unit	8-Int Add, 4-Int Mul/Div, 4-FP Add, 2-FP Mul/Div
Memory port	4-Mem Port
Branch Predictor Parameters	
Branch Predictor	Hybrid 8192-entry gshare/2048-entry Bimodal
Meta Table / BTB / RAS Size	8192 / 2048 4-way / 64
Memory Parameters	
IL1 config	64kbyte, 64byte block size, 2 way, 1 cycle latency
DL1 config	64kbyte, 64byte block size, 2 way, 1 cycle latency
UL2 config	1Mkbyte, 64byte block size, 4 way, 20 cycle latency
Mem config	300 cycle first chunk, 6 cycle inter chunk

Table 3.1: Parameter settings for the detailed architectural model into which we inject faults.

Table 3.1 lists the settings for our detailed microarchitectural simulator, which is a modified version of the out-of-order processor model from Simplescalar 3.0 for the PISA instruction set [24]. Our modifications to the original simulator include detailed modeling of rename registers and issue queues. Moreover, we mainly look at fault injection effects on three hardware structures: the physical register file, the fetch queue, and the issue queue. For faults occurring on physical registers, they do not have any architectural effect if the registers they attack are idle or belong to mis-peculated instructions, or the instruction outputs have not been written back yet. Hence, throughout all the fault injection experiments on the detailed simulator, we do not checkpoint any fault that has no architectural effect. Besides physical registers, we also simulate faults on the fetch queue. In our model, each fetch queue entry is comprised of instruction bits including opcodes, register addresses, and immediate

specifiers. Faults may occur anywhere, but only those that attack non-mispeculated instructions impact architectural state, and are recorded accordingly. Lastly, for the issue queue, we model 6 fields per entry: instruction opcode, 3 register tags (2 source and 1 destination), an immediate specifier, and a PC value. Similar to the fetch queue, faults on the issue queue (except for register tag field), affect architectural state when the entries belong to non-mispeculated instructions. For the source or destination register tags in the issue queue, corruptions corresponding to mispeculated instructions can still affect architectural state because the corruptions alter data dependence (even potentially to non-speculative instructions). Corruptions in the opcode and immediate fields of each entry behave similarly to faults on the fetch queue, while faults on the PC value affect computation for branch target addresses.

For each benchmark, we perform 3 detailed simulation runs and inject faults to the 3 hardware structures including physical register file, fetch queue, and issue queue, separately. In each run, we randomly inject faults into a single hardware structure one after another. The time before injecting another fault is determined by a uniformly distributed random variable. We also skip the initialization phase of each benchmark for fault injection.

Table 3.2 lists all the benchmarks we have studied, as well as their input datasets, numerical outputs and qualitative outputs. The first three benchmarks, G.721-D, JPEG-D, and MPEG-D, are multimedia workloads, and are taken from the Mediabench suite [25]. All three algorithms perform lossy decompression on audio, image, and video data, respectively. Then, there are three AI workloads: LBP, SVM-L, and GA. Loopy Belief Propagation (LBP) [26] is a well-known message-passing

Benchmark	Input	Numerical Output	Qualitative Output
Multimedia			
G.721-D	clinton.pcm	Decompressed Audio Datafile	Segmental Signal-to-Noise Ratio (SNRseg)
JPEG-D	lena.ppm	Decompressed Image Datafile	Peak Signal-to-Noise Ratio (PSNR)
MPEG-D	mei16v2.m2v	Decompressed Video Datafile	Peak Signal-to-Noise Ratio (PSNR)
Artificial Intelligence			
LBP SVM-L GA	WebKB [30] LIBSVM(a1a) [31] r16-0.1.in [29]	Page Belief Values Support Vector Model Thread Schedule	Web Page Class Types Test Data Class Types Best Scheduling Time
SPECInt CPU2000			
164.gzip 256.bzip2 175.vpr	test train test	Compressed File Compressed File Cell Placement	Decompressed File Decompressed File Placement Cost

Table 3.2: Input, numerical and qualitative outputs computed by our benchmarks.

algorithm for approximate inference on large Markov networks. It is widely used in coding theory and combinatorial optimization. SVM-L is the learning portion of a Support Vector Machine algorithm, called SVMlight [27]. SVM-L learns the parameters for a support vector (SV) model on a training dataset. The model is then used for classification on new datasets. The last AI program, GA, is a genetic algorithm applied to multiprocessor thread scheduling [28, 29]. Based on a task dependence graph, the algorithm searches for a thread schedule in which each thread is assigned a task, and the overall execution time across all threads is minimized. Finally, the SPECInt CPU2000 workloads we study include two lossless data compression algorithms, 164.gzip and 256.bzip2, and a place-and-route program, 175.vpr. Note in our experiments, we configure the vpr program to only perform placement.

In Table 3.2, the second column specifies the input dataset used for each benchmark. The inputs are selected so that they won't result in extremely long execution time since we have to run each fault injection trial to completion during the second phase of our two-phase simulation methodology. The third column in Table 3.2 reports the numerical outputs computed by each benchmark, and the last column reports the corresponding qualitative outputs that we have observed. For the three multimedia programs, the numerical outputs are the decompressed datafiles, either in audio, image, or video format. These decompressed datafiles, when played back to the user, are interpreted by his/her human sensation, either aural or visual. To evaluate the overall output quality experienced by users, we adopt signal-to-noise ratio (SNR), a common quality metric in signal processing. More specifically, we use segmental SNR for G.271-D to evaluate audio signal quality, and peak SNR for JPEG-D and MPEG-D for image/video signal quality evaluation. It is possible for different numerical outputs–the decompressed datafiles–to result in indistinguishable or similar experience for the user, which means similar SNR values.

Like the multimedia workloads, the AI workloads also have qualitative program outputs. In LBP, for each node in the graph, the algorithm computes a probability distribution function over the possible class types. The numerical outputs for LBP are the probability values across all the nodes. The qualitative outputs are classification answers derived from the nodes' probability distribution–the numerical outputs. In particular, LBP selects a class type for each node by choosing the most likely class indicated by the biggest probability value across the entire probability distribution. In SVM-L, the numerical outputs are the SV model parameters learned

from the training dataset. To obtain the class types we want, we run a separate SVM classifier (not listed in Table 3.2) that uses the SV model computed by SVM-L, and perform classification on a test dataset. Like LBP, for SVM-L, computing the classification answers is an extremely inexact process. Multiple numerical outputs (belief values for LBP and SV model parameters for SVM-L) can lead to the same (and hence, valid) classification answer. Lastly, for GA, the numerical output is the thread schedule the program computes. In most cases, the computed thread schedule is an approximation to the optimal answer. This is because the program has a solution space which is too big to explore exhaustively in practice. Thus the algorithm adopts a heuristic approach and produces a relatively good answer. The thread scheduling cost can reflect how good the computed answer is. Thus it is possible that many thread schedules are adequate with acceptable cost. And any one of these good enough answers represent a valid output from the user's perspective. In Table 3.2, we use the thread scheduling cost as the algorithm's qualitative output.

More interestingly, even the three SPEC programs also allow multiple valid outputs, although they are traditionally considered as exact computations. Both the data compression programs, 164.gzip and 256.bzip2, are lossless algorithms and have to compute exact decompressed datafiles. Nevertheless, there is flexibility in how datafiles are compressed even though the compression algorithms themselves are exact. In other words, the compressed file, which is a numerical output, could be different and still lead to exactly the same datafile after decompression. We consider the decompressed file as the programs' qualitative result. Valid execution is allowed to produce a different compressed file but the same original datafile must

be re-produced after decompression. Hence we use the compression ratio to reflect compression efficiency of each valid execution in our experiments. The vpr benchmark tries to find a cell block placement for a chip design. Like GA, vpr's algorithm is heuristic-based since finding an optimal placement (one that minimizes interconnect distance) is intractable. Therefore, multiple cell block placements are valid. To evaluate valid execution, we use a consistency check provided by the vpr code itself. This function first checks whether a given placement for all the cell blocks is valid (*i.e.*, doesnot violate any design rules), and then if it is valid, the checker computes the cost for the given placement. We use the computed cost as the output quality for vpr.

All the numerical and qualitative outputs listed in Table 3.2 are used in the second phase of our fault injection experiments to evaluate program execution correctness–under both architecture and application level, respectively. We will present our results in the next section.

3.2.2 Fault Susceptibility Result

In the first phase of our fault injection experiments, we inject faults to the detailed simulator and checkpoint the affected state. In all, our fault injection campaign performs 156,205 fault injections–52,555 for physical register file, 52,229 for fetch buffer, and 51,421 for issue queue across all the benchmarks. However, not all faults result in visible architectural effects–only a small portion of the injected faults affect program architectural state. Many faults are masked at the microarchi-

tecture level. As discussed by Mukherjee *et al* [18], microarchitecture-level masking is mainly due to faults that attack idle hardware resources, or hardware resources occupied by mispeculated instructions.

Figure 3.1 breaks down all the fault injection experiments by whether they are architecturally visible or not. In Figure 3.1, fault injection experiments on each benchmark are grouped together, with each group consisting of three bars representing experiments on the physical register file, fetch queue, and issue queue, labeled "R," "F," and "I,"respectively. The last 3 groups of bars report the average across each category of benchmarks, *i.e.*, the multimedia, AI, and SPEC benchmarks, respectively. We can see that the degree of masking at the microarchitecture level varies considerably across different benchmarks and hardware structures. But on average, across all the hardware structures and all the benchmarks, only 17.3% of injected faults (27,067 out of 156,205) become architecturally visible. Among all the structures, the fetch queue exhibits the most sensitivity to faults, with 22.6% faults being architecturally visible; while the register file and issue queue are less sensitive, with 12.1% and 17.3% faults resulting in visible architectural effects, respectively. Faults that are masked at the microarchitecture level produce exact program outputs that are correct under both architecture- and application-level correctness definitions.

Next, we evaluate program output correctness to further examine the architecturally visible faults. Figure 3.2 breaks down the outcome of all fault injection experiments that have visible architectural effects. Note those outcomes are obtained with our functional simulator and each experiment is run to completion if

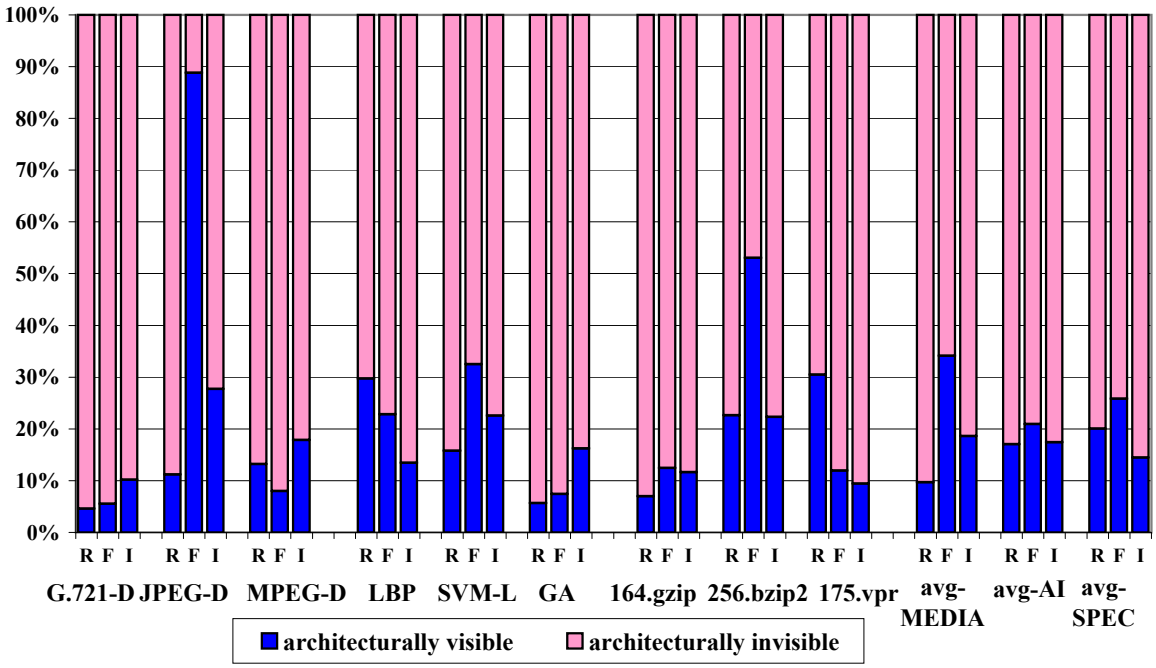

Figure 3.1: Breakdown of fault injections on architectural visibility.

possible. Figure 3.2 is presented in the same way as Figure 3.1, except that each bar in Figure 3.2 is now broken down into 6 categories. The first category, labeled "Architecture," indicates the portion of experiments which have program outputs that pass architecture-level correctness (definition II in Section 3.1.2). Naturally, these outputs are also correct at the application level. The next two categories, labeled "Application-High" and "Application-Good," represent the portions of experiments which have incorrect outputs under architecture-level correctness, but are acceptable under application-level correctness (definition III and IV in Section 3.1.2, respectively). In our experiments, we set the thresholds for "Application-High" and "Application-Good" to be 1% and 5% error, respectively, when comparing programs' qualitative outputs to fault-free execution. Exceptions are JPEG-D and MEPG-D, for which "Application-High" means PSNR of the output is greater than 90dB, while

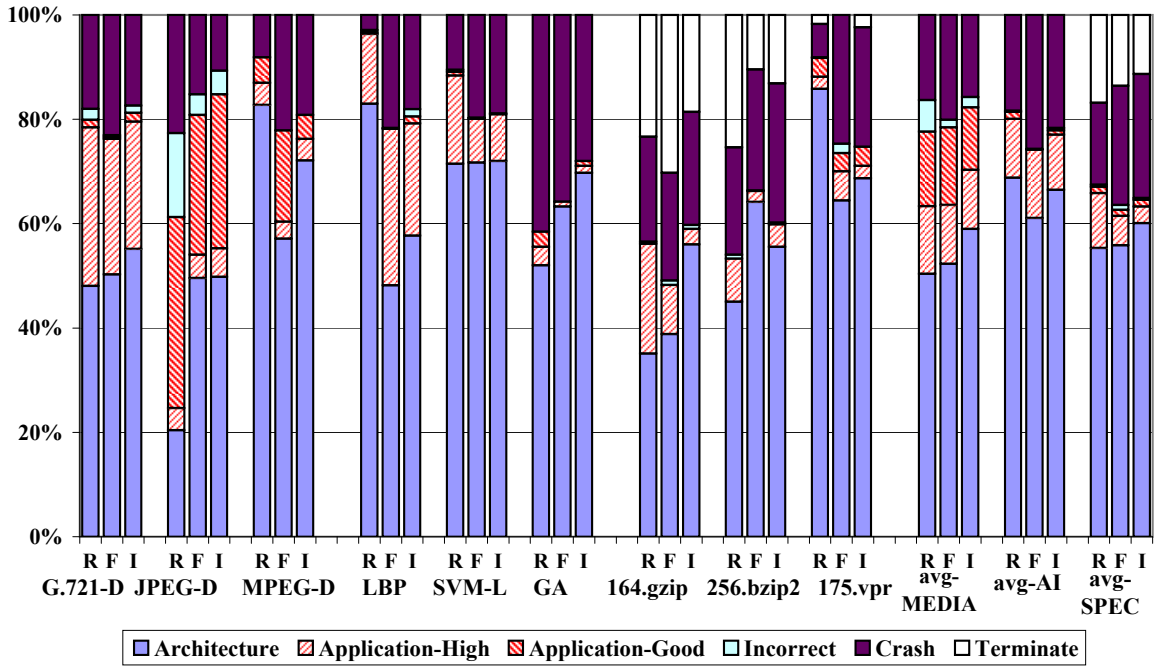

Figure 3.2: Breakdown of program outcomes for architecturally visible fault injections.

"Application-Good" means PSNR is between 90dB and 50dB, when compared to the original output. The next category, labeled "Incorrect", reports those program outcomes that are incorrect under both architecture- and application-level correctness. Finally, the last two categories indicate experiments that cannot complete during functional simulation: the category labeled "Crash" reports experiments which have failed because of exception or hardware lockup, while the category labeled "Terminate" reports trials in which programs have detected errors themselves and decided to exit early. Note in our simulations, program lockups are detected via expiration of watchdog timers that are set at the beginning of major loops in each benchmark.

From Figure 3.2, we see a large portion of architecturally visible faults lead to correct program outputs under architecture-level correctness (*i.e.*, the "Architecture" components). The last 3 groups of bars in Figure 3.2 show on average

that architecture-level correctness is achieved in 50.4% to 60.0% of program outputs across the 3 hardware structures for the multimedia and SPEC benchmarks, and in 61.0% to 68.8% for the AI benchmarks. Those faults only affect architectural state that are unnecessary for maintaining numerical integrity in our computations, and become architecturally masked as program's execution proceeds. In our benchmarks, the primary source of architecture-level masking is logical and inequality instructions. These instructions seldom change their computation results despite data corruptions to their input operands; thus, they are highly fault resilient. Similarly, faults may also be masked by shift or bitwise operations. Other (less significant) sources of architecture-level masking include dynamically dead code, NOP instructions, and Y-branches [19]. Additionally, we find memory operations also contribute to fault masking, such as partial stores in which corruptions occur in part of the data which is not stored to the memory, thus they have no impact on memory integrity; or load instructions which are supposed to load data which are common in memory (for example, zero), therefore corrupted load addresses may point to other locations which fortunately contain the same data value. Both microarchitecture- and architecture-level masking have been previously observed by other fault susceptibility studies [18, 32, 19].

For the remaining portions of experiments with final results produced, faults are not masked at both microarchitecture and architecture levels, thus they do not lead to numerically correct outputs. These faulty outcomes have traditionally been considered *incorrect* under architecture-level correctness. In our study across all benchmarks and all hardware structures, on average about 41.2% of architecturally

visible fault injections are architecturally incorrect. However, we find among those architecturally incorrect outcomes, a significant portion still yield fairly good solution quality. This is particularly true for soft computations, the multimedia and AI benchmarks in our study. While these program outputs are incorrect numerically (*i.e.*, they are incorrect at the architecture level), they are completely acceptable from user's standpoint (*i.e.*, they are correct at the application level). Overall, 45.8% of architecturally incorrect faults in our soft computations are tolerable under application-level correctness.

More interestingly, not only soft computations, but also traditionally exact computations, *i.e.*, SPEC benchmarks in our study, exhibit enhanced fault resilience at the application level. As the last group of bars in Figure 3.2 shows, for the SPEC benchmarks, overall 17.6% of additionally acceptable faults are achieved across all hardware structures. Although our results in Figure 3.2 show that SPEC benchmarks offer less additional fault resilience at the application level compared to soft computations, they still indicate that application-level correctness can generally help to enhance program fault tolerance.

3.3 Sources of Fault Tolerance

Section 3.2.2 demonstrates that a significant portion of faults that lead to numerically incorrect results are in fact perfectly acceptable to the user, and are no longer perilous at the application level. There are many factors that enable such additional error resilience. In our study, we find in addition to the existence of

qualitative program outputs that have been discussed in Section 3.1.1, there are also algorithmic properties that contribute to application-level fault tolerance. We have identified the following properties which are common in applications, especially in soft computations.

I. **Redundancy**. Computations that are iterative or that exhibit reduced precision (see below) often contain some degree of redundancy. Unlike dead code, these redundant computations contribute to the application result, but may not improve answer quality appreciably. Programs with redundant computations are more error resilient because the redundancy can mask faults.

II. **Adaptivity**. Many soft computing algorithms are already designed to deal with errors. This is particularly common in algorithms that compute on noisy or probabilistic data. Such soft computations include code to detect certain forms of error, and adapt the computation accordingly. Therefore, they are naturally error resilient.

III. **Reduced Precision**. Programs often have precision requirements that are lower than the datatypes supported by the programming environment or hardware architecture. Such computations are resilient to errors that modify data values within the precision tolerance.

IV. **Efficiency-aimed Computation**. Some codes are designed for the goal of improving program efficiency, and do not closely relate to result accuracy. Corruptions on that kind of computation are tolerable in the sense that they only impact program performance, not output correctness.

V. **Sub-Independence**. Some algorithms are comprised of separate parts of computations which do not depend on each other. Even if faults occur on some parts of these computations, other parts could still execute normally, thus the final results may still be acceptable if the overall disturbance is minor.

We find that these algorithmic sources of fault tolerance, together with qualitative program outputs, not only help to mask many architecturally unacceptable faults so that they are now acceptable at the application level, but also contribute to the feasibility of lightweight recovery techniques, which will be introduced next in Chapter 4.

Chapter 4

Fault Recovery by Exploiting Application-level Correctness

This chapter presents our work on fault recovery by exploiting application-level correctness. First, Section 4.1 proposes a light-weight recovery technique by only checkpointing the program counter, register file and stack, periodically. Then Section 4.2 extends the technique by identifying "soft" state in programs and excluding them from checkpoints. In Section 4.3, both performance and cost of such recovery mechanisms are reported.

4.1 Lightweight Fault Recovery

As discussed in Section 3.2.2, many architecturally incorrect faults are acceptable when examined at the application level. However, even after considering application-level correctness, a large portion of faults still lead to incorrect program execution–*i.e.*, the "Incorrect," "Crash," and "Terminate" components in Figure 3.2. Of these, by far the "Crash" component is the most significant portion. In all but three bars (the "R" and "F" bars for gzip, and the "R" bar for bzip2), the "Crash" component dominates. From Figure 3.2, we can see that across all benchmarks and all hardware structures, on average crashes account for 80.8% of faults that are incorrect at both the architecture and application levels. Techniques that can address crashes will have a large impact on fault tolerance as fault rates keep increasing in

the future. Additionally, the "Terminate" category of trials is similar to the "Crash" category in the sense that both exit prematurely and have no or only part of outputs generated. The main difference between the two categories lies in whether the application itself has been implemented to detect certain forms of errors and exits early accordingly. Therefore, techniques that can address crashes may also be able to deal with faults in the "Terminate" category as well. In this work, we only look at recovering faults leading to program crashes; more work needs to be done for specific applications in order to deal with program "Termination."

In our mechanism, crashes manifest themselves as either exceptions or program lockups. Program lockups are detected via expiration of watchdog timers that are set at the beginning of major loops in each program. To recover from faults, we find that lightweight checkpoints can be effective thanks to the existence of algorithmic sources of fault tolerance, as well as qualitative program outputs discussed in Section 3.3. Next, we will discuss our lightweight recovery technique in Section 4.1.1, and then present our results in Section 4.1.2.

4.1.1 Lightweight Recovery Mechanism

Once a fault occurs and before it is detected, corruptions may propagate anywhere in the computation. While recovering all the modified data is necessary for architecture-level correctness, it is overly conservative for application-level correctness because soft computations, as a result of the algorithmic properties discussed in Section 3.3, are resilient to data corruptions, and program outputs do not need

to be numerically perfect.

To reduce checkpoint overhead, in our mechanism, we only checkpoint the portion of state that is necessary for restarting program execution. More specifically, we find that in most cases, only a valid PC, architected register file, and program stack are enough for successful recovery. Once a crash is detected, we roll back program state including PC, register file, and stack, to the nearest checkpoint, and then restart the program–we do not touch other state such as program text, static data, or heap during rollback. Our checkpoints are instrumented at the beginning of the main controlling loops in our benchmarks.

Note, in our lightweight recovery mechanism, we checkpoint program state heuristically, and it is possible that some state necessary for program restart are not saved in our checkpoint, thus some crashes may not be recoverable. However, the main advantage of our technique is that it is very cheap compared to a perfect fault tolerance technique which requires us to identify all the state that are necessary to recover all possible crashes, and save them frequently enough. In our technique, across all the benchmarks, the average checkpoint size is 3 Kbytes, and only accounts for 0.4% of the total program state. Furthermore, our checkpoints are incurred very infrequently since we only instrument checkpoints at the beginning of the main controlling loops in each program: there are at least 400,000 instructions between consecutive checkpoints. Moreover, as we can see in the next section, our recovery mechanism is not only lightweight, but also very effective and can successfully recover a significant portion of crashes in many cases.

4.1.2 Lightweight Recovery Results

We perform recovery for all crash trials ("Crash" component in Figure 3.2), using the functional simulator which is instrumented with our lightweight checkpoint mechanism. Each time a program crash is detected, as described in Section 4.1.1, we rollback to the nearest checkpoint, restart execution, and try to run the benchmark to completion. If the program doesnot crash again, we evaluate its outputs under both architecture- and application-level correctness, just as we have done in Section 3.2.2. Figure 4.1 breaks down the outcome of our recovery experiments. Each bar, representing experiments on one hardware structure for one benchmark, is broken down into the same categories as Figure 3.2, except that there is no "Terminate" category since none of our recovery experiments end in early program exit. The last 3 groups of bars in Figure 4.1 report the average breakdowns for the multimedia, AI, and SPEC benchmarks, respectively.

First, Figure 4.1 shows that our lightweight recovery technique is helpful even with traditional numerical correctness. A number of program crashes can be recovered successfully and generate exact outputs under architecture-level correctness (*i.e.*, the "Architecture" components). For soft computations, the multimedia and AI benchmarks, on average about 3.8% to 17.7% of recoveries are architecturally correct, while for SPEC programs, architecture-level correctness is achieved on average from 21.5% to 30.8% of all recoveries. By examining these experiments more carefully, we find that in most cases, faults have not corrupted uncheckpointed state between the time the fault occurs and its subsequent crash. Thus, our checkpoint

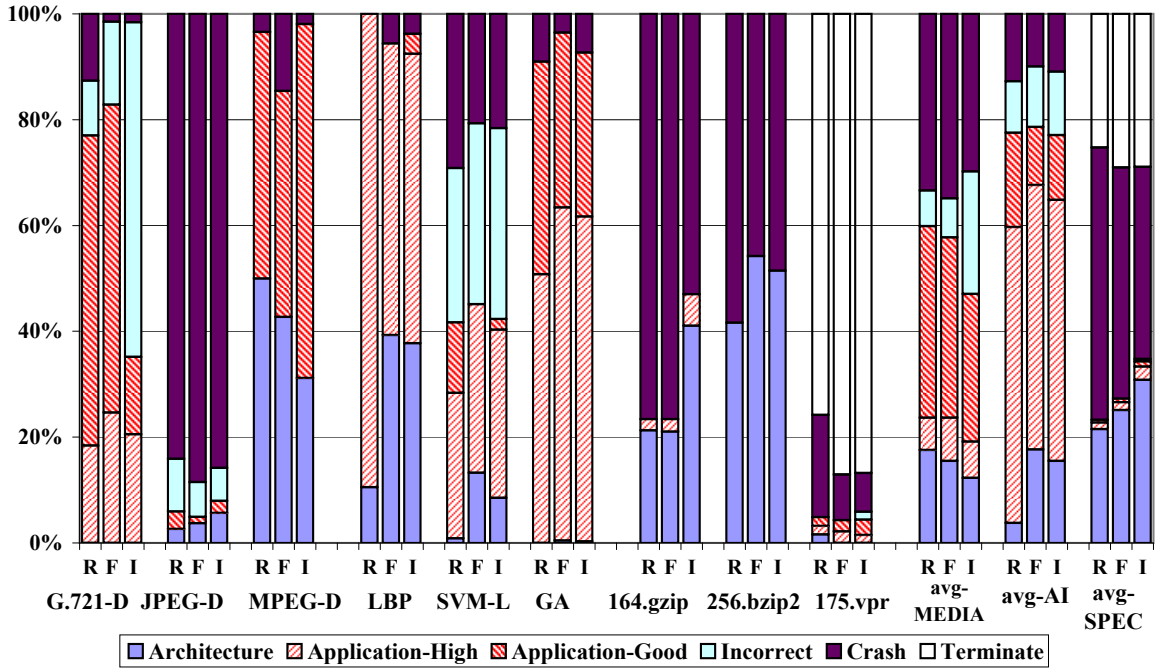

Figure 4.1: Breakdown of program outcomes for lightweight recovery of crashes.

can correct corruptions on state that have been checkpointed, and allows program to complete with numerically perfect outputs.

Furthermore, our lightweight recovery technique can be even more helpful under application-level correctness. Figure 4.1 shows that a significant number of additional crashes can be recovered correctly with the application-level correctness definitions (*i.e.*, the "Application-High" and "Application-Good" components). This is especially true for soft computations. Look at the average bars for the multimedia and AI benchmarks in Figure 4.1. On average, about 34.8% to 73.8% of recoveries are correct under application-level correctness, although their numerical outputs are incorrect under architecture-level correctness. Moreover, across all hardware structures and all soft computations, an additional 52.6% of recoveries are acceptable under application-level correctness. Combining both numerically correct

and qualitatively correct recoveries, for soft computations, our lightweight recovery technique allows on average about 66.3% of all crashes to complete correctly. In addition, combining results from Figure 3.2, with our definition of application-level correctness and lightweight recovery mechanism, for soft computations, about 92.4% of all architecturally visible faults are acceptable. That is, they still lead to correct outputs at either the architecture or application level. However, for the SPEC benchmarks, our lightweight recovery mechanism does not perform as well as for soft computations. As shown in the last group of bars in Figure 4.1, at most an additional 2.5% of all recoveries are acceptable under application-level correctness, on top of the numerically correct recoveries. This shows again that soft computations are resilient to errors while exact computations like the SPEC benchmarks are not.

In summary, application-level correctness can enhance program fault tolerance even when infrequently checkpointing a small amount of program state. The benefit is more significant for programs which have multiple valid outputs as well as algorithmic properties for fault resilience (*i.e.*, soft computations).

4.2 Soft Checkpoint Mechanism

As we have presented in Section 3.2.2, programs can tolerate more errors under application-level correctness compared to the traditional architectural-level correctness. Furthermore, Section 4.1.1 shows that in case of fatal faults, only restoring a small set of program state can help program to restart and still generate

acceptable outputs. However, our lightweight recovery mechanism is not perfect in that it is not able to recover all the crashes even for soft computations (Section 4.1.2). The problem is mainly caused by the fact that our lightweight checkpoints are too simple to contain all the state necessary for program's normal execution–*i.e.*, faulty corruption of some state that are not included in the checkpoints can prevent program from generating acceptable outputs. Thus, in order to achieve successful recovery on most crashes, it is crucial to identify the portion of state that have to be exact for proper program execution, or on the contrary, the portion of state that are highly resilient to errors–omitting them from checkpoints does not sacrifice program correctness at the application level. In this work, we call the latter "soft state", and refer to all other program state as "hard state". For many applications such as the soft computations in our study, soft state only constitute part of the whole program state, thus rendering the possibility of selectively checkpointing hard state to achieve smaller checkpoint size and cheaper recovery cost compared to traditional full checkpointing. In addition, to maintain recovery performance via soft checkpointing, hard state have to be identified throughout the whole program, which requires complete and accurate code analysis. In our experiments, we bypass such high requirement by identifying soft state and assuming all other program state as hard state. Thus, incompleteness in our analysis only affects checkpointing cost since less soft state are to be omitted from checkpoints, while the recovery performance is not influenced.

In this section, we mainly present our study on two key issues in implementing such soft checkpointing technique. First, Section 4.2.1 discusses how to identify

soft program state. Then, Section 4.2.2 reports our checkpointing and recovery mechanism.

4.2.1 Soft Program State

Soft state are those that can be inexact–in other words, their numerical values can be different from fault-free execution–while programs can still generate valid outputs. Once soft state are identified, checkpoint cost–either its size or performance overhead, as well as fault recovery cost–can be improved by excluding the soft state from checkpoints. Note the soft state we discuss here are not dead or read-only data which have been explored in previous work such as [33, 34]. Unlike dead data which comprise values that are irrelevant to program execution, or read-only data which contain the same values since the last checkpoint, errors in soft state do change program execution–usually they may cause program to produce numerically different outputs. Those errors are intolerable under traditional architectural-level correctness. However, at higher levels of abstraction, *i.e.* at the application level, they may be tolerated if the corresponding outputs still fulfill user's requirements.

To identify soft program state, generally there exist two approaches. The first approach comes from the observation that most soft state directly relate to soft program outputs, which are also referred to as qualitative outputs in Chapter 3 due to the qualitative nature of the results. Therefore, given information on soft program outputs, backward dependence analysis can be applied to collect all possible data that contribute to soft outputs. Although it is possible some state are error-

resilient even if they do not directly compute soft outputs, such cases are not common and most soft state can be identified in this fashion. The other approach, which is easier to implement, relies on programmer's knowledge to inspect the code and identify data structures associated with soft outputs. Such approach is similar to the technique proposed by [33, 34] in which programmer gives directions on excluding or including memory regions to checkpoints. One concern with such approach is that programmer may mistakenly mark non-error-resilient data structures as soft. However, such mistakes can be avoided if the programmer is conservative in selecting soft data. In our experiments, we take the second method and our results presented in Section 4.3 show our technique is effective in checkpointing necessary state for recovery.

In addition, in selecting soft state, we only examine heap data structures since they are the main sources of soft program state across our benchmarks. Other memory state including static data and stack are ignored in our analysis, though the PC, register file, and stack are automatically included in the checkpoints (similar to our lightweight checkpointing mechanism).

4.2.2 Soft Recovery Mechanism

As we have mentioned, our soft checkpoints include the PC, register file, and stack–we save them fully at each checkpoint since their size is usually very small and thus this introduces little overhead. In addition, our soft checkpoints also include hard state existing in other parts of memory such as static data and heap. Unlike

register file or stack, these memory regions generally involve huge amounts of data. Hence, instead of using full checkpoints, we employ incremental checkpointing. Incremental checkpointing is traditionally one of the most efficient ways to checkpoint program state. It works by maintaining a list of objects that have been updated since the most recent checkpoint. To establish a new checkpoint, only the dirty objects on the list are saved, thus eliminating redundant copies. After each checkpoint, the list is cleared so that it can be used to track dirty objects for the next checkpoint. In our mechanism, we only maintain the list to keep track of updates on hard state.

Although ideally only updates on hard state are to be checkpointed, the granularity at which modifications of hard state are tracked impacts the size of the checkpoints–usually the coarser granularity updates on hard state are examined, the bigger the resulting checkpoints are, as well as the related performance overhead. However, to achieve finer granularity, more hardware are required to maintain the list of dirty objects. In our study, we track modifications of hard state at page granularity, thus the hardware in TLBs for tracking dirty pages can be utilized for our purpose. While this is cost effective, it also incurs some overhead since most data objects are smaller than a page. We will discuss more about this later.

To distinguish modifications on hard state, we identify store instructions that write to the data structures containing soft state. We refer to these store instructions as "soft stores". All other stores not identified as soft stores are conservatively assumed to update to hard state, and are identified as hard stores. The data structures for soft state are collected by examining the code and picking those structures

that are associated with soft outputs (the soft outputs for our benchmarks are listed in Table 3.2). Once soft stores have been identified, we mark them in our benchmarks' binaries so that they can be recognized at runtime. In our experiments, this is achieved by creating new instructions with unused opcodes in the simulated instruction set, and then replacing the original soft stores. As we will explain in the next section, this enables our soft recovery technique to omit dirty pages which are only touched by soft stores, thus saving checkpoint size and overhead.

4.3 Soft Recovery Results

In our experiments, we modified our detailed out-of-order simulator from Chapter 3 to support incremental checkpointing. As our simulator does not model TLBs which are normally used for incremental checkpointing, we track dirty pages in the simulator itself instead. For every executed store instruction, our simulator first checks whether it is a soft store or not; if it is not, the simulator then observes which memory page the store writes to (assuming 4,096 bytes per memory page). Once a memory page is found to be modified for the first time, the simulator appends its page number to a modified page list. At each checkpoint, the modified page list is traversed and the recorded dirty pages are checkpointed using a copy function. The copy function also checkpoints the PC, register file, and stack. In addition, the copy function is executed at runtime and fully simulated, thus the performance cost it incurs can be evaluated.

This section presents the results by applying our soft recovery mechanism to

Bench	Total Pages	Dirty Pages	Dirty Pages	Dirty Blocks	Dirty Hard Blocks
G.721-D	6	3 (0.50)	3 (1.0)	13	13 (1.0)
JPEG-D	222	14 (0.063)	7 (0.50)	369	146 (0.40)
MPEG-D	244	8 (0.033)	2 (0.25)	216	20 (0.093)
LBP	1906	1444 (0.76)	1 (6.9e-4)	33633	1 (3.0e-5)
SVM-L	237	34 (0.14)	19 (0.56)	578	359 (0.62)
GA	11779	46 (0.039)	38 (0.83)	349	46 (0.13)

Table 4.1: Checkpoint size statistics. "Total Pages" reports the total number of memory pages allocated in each benchmark. "Dirty Pages" reports the average number of pages that are updated during one checkpoint period after program initiation, which corresponds to the traditional incremental checkpoint size. (The numbers in parentheses are fractions of the total pages.) "Dirty Hard Pages" reports the average number of pages in which non-soft blocks are updated during each checkpoint period, corresponding to our soft-checkpoint size. And the numbers in parentheses are fractions of the dirty pages. The last two columns, "Dirty Blocks" and "Dirty Hard Blocks", report the average size of updated memory–and the non-soft part, separately–during one checkpoint period by the number of memory blocks. For the latter, the numbers in parentheses are fractions of the dirty blocks.

the soft computations in our study–*i.e.*, multimedia and AI workloads. We first report the soft checkpoint cost, including checkpoint size and the impact of checkpointing on program execution time (Section 4.3.1). Then, we demonstrate how effective our soft checkpointing technique is in recovering program crashes. We also discuss some observations from our experiments (Section 4.3.2).

4.3.1 Soft Checkpoint Cost

We evaluate two aspects of checkpoint cost–the size of checkpoints, as well as the performance overhead they cause on each program. In order to compare our soft checkpointing with the traditional incremental checkpointing, we first run each benchmark on our detailed out-of-order simulator to acquire checkpoints assuming no soft store instructions; then, we run each benchmark again and acquire check-

points omitting modifications by soft stores. Note the checkpoint locations are the same as those described in Section 4.1.1.

Table 4.1 presents the average size of checkpoints for each soft computation program in our study. In the table, the column labeled "Total Pages" reports the total number of memory pages allocated in each benchmark, excluding code segment and stack. The column labeled "Dirty Pages" shows the average number of pages that have been updated during one checkpoint period after program initiation (the numbers in parentheses are the fractions over the total pages)–these numbers reflect the size of traditional incremental checkpointing. On average, only 25.6% of all the pages are modified between two checkpoints. In comparison to traditional checkpointing, the column labeled "Dirty Hard Pages" reports the average number of pages which have been modified by hard store instructions during each checkpoint period–the numbers in parentheses are the fractions over the dirty pages. The table shows for most of the soft computations, only saving memory pages that contain updated hard data reduces checkpoint size significantly–for some benchmarks such as LBP, almost all of the updated data are error-resilient, which results in extremely small checkpoints compared to traditional incremental checkpoint. But for other benchmark such as G.721-D, updated hard data are distributed through all of the memory pages, thus there is no reduction on checkpoint size after considering data softness. On average, soft checkpoints are about 52.3% of the size of conventional incremental checkpoints.

Moreover, if we checkpoint at finer level of granularity–*i.e.*, saving updated data by memory blocks, our soft checkpointing mechanism can save even more. In

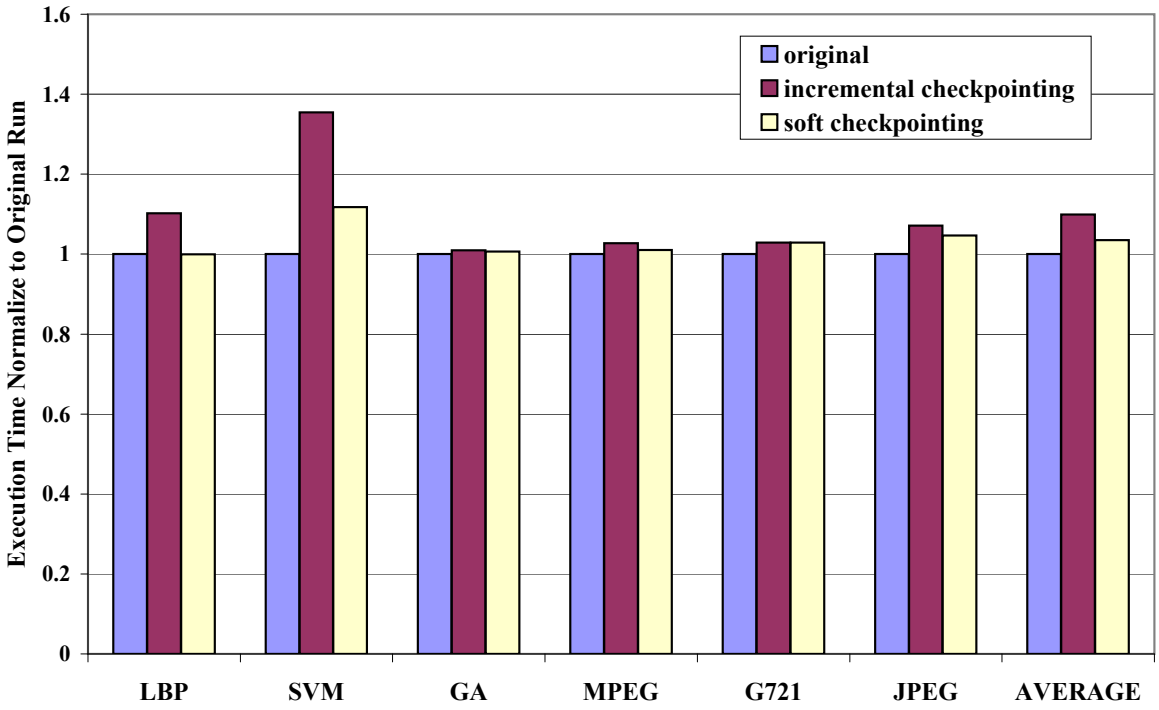

Figure 4.2: Program execution time with traditional incremental checkpointing or soft checkpointing technique.

Table 4.1, the columns labeled "Dirty Blocks" and "Dirty Hard Blocks" present the average number of memory blocks that require copying under traditional incremental checkpointing and our soft checkpointing technique, respectively. (For the column labeled 'Dirty Hard Blocks", the numbers in parentheses are the fractions over the dirty blocks.) Compared to checkpointing by memory pages, for most of the applications, checkpointing by memory blocks results in smaller checkpoint size. On average, the space required by soft checkpoints is only about 37.4% of what incremental checkpoints need. However, as we have discussed, using smaller checkpoint granularity incurs more hardware overhead, such as bits for recording update status.

Because soft checkpointing requires much less data to be saved, its runtime

overhead for saving checkpoints is also much smaller. Figure 4.2 shows our results on performance overhead. For each benchmark, which corresponds to one group of bars in the graph, we report the execution time (cycles) with traditional incremental checkpointing or our soft checkpointing technique, normalized to the original (no checkpointing) program execution. The last group of bars exhibits the average results across all the benchmarks. The graph shows for each application, comparing the traditional and our checkpointing techniques, their runtime overhead is approximately consistent with their checkpoint size reported in Table 4.1. On average, the traditional incremental checkpointing incurs about 10.5% runtime overhead on program execution, while our soft checkpointing technique only causes about 3.3% overhead, which shows the benefit from the smaller size of our soft checkpoints.

4.3.2 Soft Recovery Performance

Not only does soft checkpointing cost less in terms of both space requirement and performance overhead, it also works effectively in recovering program crashes. Recall in our mechanism, upon program crashes, we restore program state with the latest soft checkpoint, and then resume program execution. We apply soft checkpointing and recovery to all the program crashes from our fault injection experiments, which are reported in Section 3.2.2. Figure 4.3 breaks down the program outcomes of our fault recovery experiments. The results are presented in the same manner as Figure 4.1 except we use soft checkpointing for recovery instead of lightweight checkpointing. Also, we only evaluate soft computations here.

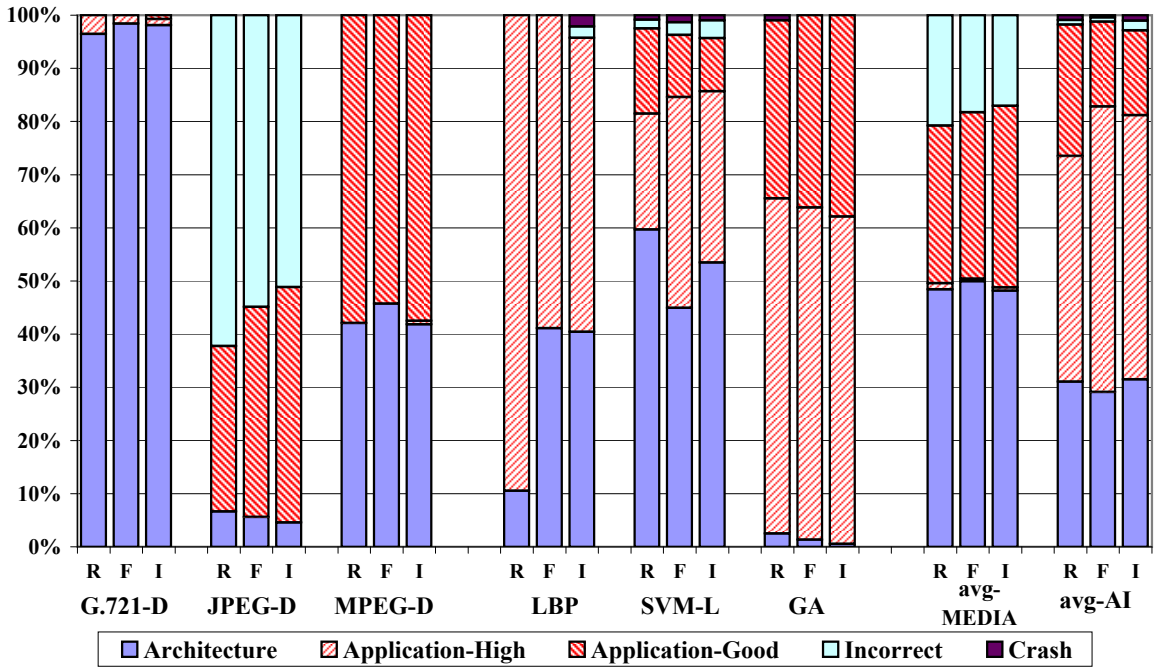

Figure 4.3: Program outcomes breakdown for soft recovery of crashes.

Figure 4.3 shows with soft recovery, almost all the program crashes are recovered and programs resume their execution until the end of these execution. The portions of "Crash" after recovery are reduced greatly–at most 1.1% across the benchmarks. Especially for multimedia applications, all the experiments can pick up their execution until producing outputs after restoring soft checkpoint. In addition, among all the experiments generating outputs, the majority are counted as correct under either architectural or application-level correctness. Look at the average bars listed in the last two groups in Figure 4.3. On average, about 79.2% to 98.8% of recoveries are correct at the application level. In particular, about 81.3% of all the crashes in multimedia programs, or 98.0% for AI programs, are successfully recovered to either architecturally correct or qualitatively correct, resulting in about 89.7% for successful recoveries across all hardware structures and all soft compu-

tations. Compared with lightweight recovery reported in Figure 4.1, which has a 66.3% recovery rate under application-level correctness across all the experiments, soft checkpointing successfully recovers an additional 23.4% of program crashes, while its performance overhead is very low–about 3.3% as reported in Figure 4.2.

Although soft checkpointing appears very effective in recovery, it is still not perfect in recovering all program crashes. In particular, Figure 4 shows a very small number of recoveries (1.1% or less for all benchmarks) result in a second crash. For these cases, we find checkpoints used for restoring program state contain corrupted data. In our experiments, checkpoints are taken at the beginning of main loop iterations. While recovery is triggered at program crashes–*i.e.*, when an exception or lockup is detected–a checkpoint may be taken after a fault occurs but before it is exposed as a program crash, which causes the checkpoint to be corrupted. In addition to crashes the second time around, some recoveries lead to incorrect outcomes. Figure 4.3 shows that averaged over all three hardware structures, about 18.7% and 1.1% of recoveries result in incorrect outcomes for the multimedia and AI benchmarks, respectively. Although most of these incorrect outcomes still exhibit good solution quality, this result indicates soft checkpointing alone does not guarantee correct execution. In some cases, faults may corrupt some unprotected soft state so that the solution quality is degraded sufficiently to make the result unacceptable.

In addition to the problems of corrupted checkpoints and unacceptable solution quality–the former can also appear in other checkpointing mechanisms, there exist other limitations in applying soft checkpointing. First, only programs computing

on soft data can benefit from such a mechanism. In our experiments, we mainly study multimedia and AI workloads. We plan to study more application domains in the future. Secondly, we find soft data do not tolerate errors equally. In other words, some data may be extremely resilient to errors while some other data can only allow a small deviation from exact execution. Such softness discrepancy affects the effectiveness of soft checkpoint in recovery, and possibly makes it more complex in determining what to checkpoint.

Chapter 5

Fault Detection with Value Prediction

In previous chapters (Chapter 3 and Chapter 4), we have discussed our study on program state redundancy at the application level. Our experimental results show that programs–especially soft computations–can tolerate more errors if their final state–their outputs–are interpreted from the user's standpoint. We also implement a lightweight fault recovery technique by only checkpointing a small portion of program state. In addition to our work on program state redundancy, we now explore another kind of redundancy which is also inherent to program execution–value predictability. Value predictability is found to exist in instruction and data streams, and has been widely studied to break true data dependence and improve program ILP. In our work, we find it can also be exploited to improve program reliability, more specifically, through low-cost fault detection.

In this chapter, we first introduce how we apply value prediction for fault detection in Section 5.1. Here, we discuss our study on characterizing instructions' vulnerability to faults as well as our methods in selecting instructions for fault protection. Then, Section 5.2 describes our experimental methodology, and reports both the reliability and performance results. Next, Section 5.3 compares our technique against fault screener [12], another technique that exploits the inherent redundancy in programs to improve fault tolerance.

5.1 Reducing Error Rate with Value Prediction

This section discusses in detail how value prediction is applied to reduce error rate. First, Section 5.1.1 introduces the main value predictors that have been explored in the literature. Then, Section 5.1.2 discusses how we use value predictors to check instruction results. Next, Section 5.1.3 briefly describes fault recovery. Lastly, Section 5.1.4 quantifies instruction's vulnerability to faults and proposes selectively predicting instructions to mitigate performance loss.

5.1.1 Value Predictor Background

Value prediction has been widely studied to improve program performance by breaking data dependence–instruction result is predicted and fed to dependent instructions, which can then proceed speculatively, thus program ILP is enhanced. To make a prediction, an instruction's past results are recorded, and its next result is predicted based on the recorded outcomes. So far, several hardware predictors have been proposed for value prediction including last value prediction, stride prediction, context prediction and hybrid prediction [35]. Each predictor differs in how it encodes instruction's past results and makes prediction accordingly.

Last Value Predictor Last value predictor works for instructions which produce the same results for consecutive instances. It stores an instruction's result produced at the most recent time, and predicts the same value for the next time the same instruction is encountered. Figure 5.1 shows the scheme of last value prediction. In the predictor, the Value History Table (VHT) keeps the instructions' up-to-date

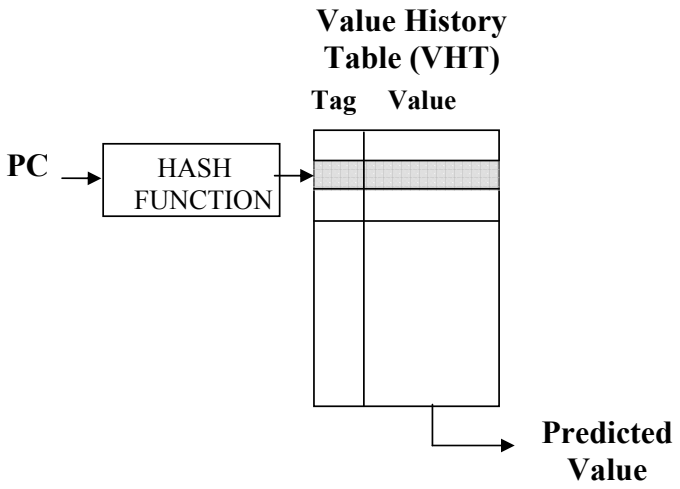

Figure 5.1: Diagram of Last Value Predictor.

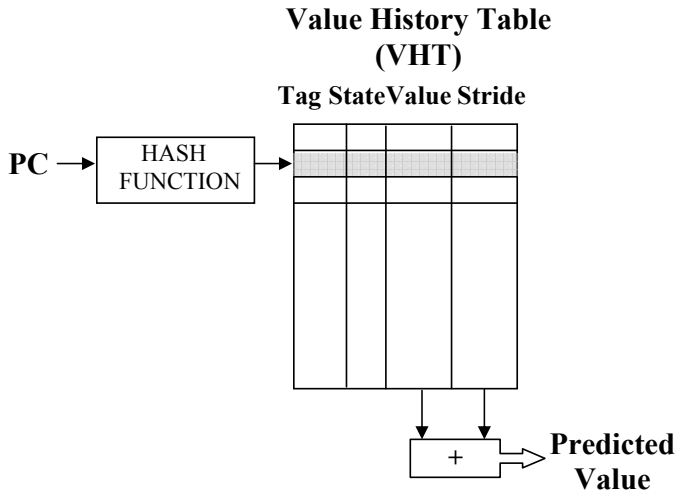

Figure 5.2: Diagram of Last Value Predictor.

outcomes, which are stored in the "Value" field of each entry. The other field in each VHT entry, labeled "Tag", is used to identify the instruction mapped to that entry. During each prediction, the program counter (PC) of the instruction to be predicted is used after a HASH function to select an appropriate VHT entry, then the data stored in the entry's "Value" field is adopted as the predicted value. After the instruction has finished its computation, its newest result is used to update the "Value" field of the corresponding VHT entry.

Stride Predictor Stride predictor captures the pattern of some instructions' computation that the results of their consecutive instances differ by a constant value. Figure 5.2 shows the basic scheme of a stride predictor. Similar to last value predictor, stride predictor also contains a VHT which contains instructions' most recent outcomes in field labeled "Value" as well a "Tag" field for instruction identification. In addition, each VHT entry also contains a "Stride" field for storing the difference between the stored outcome and its precedent, and a "State" for indicating the status for making prediction with the stride. When a new instance of an instruction is executed, the difference between the new value and the last-value field is written into the "Stride" field, and the new value itself is written into the "Value" field. If the same stride value is computed twice in a row, the corresponding "State" field is marked as "steady", and the predictor predicts the instruction's next value as the sum of the "Value" and "Stride" fields. When a computed stride differs from the previously computed stride, its "State" field is marked as transient, and the predictor stops making predictions until the stride repeats again.

Context-based Predictor Context predictor can capture more complex patterns of instruction computation than last value predictor and stride predictor. Figure 5.2 illustrates the basic scheme of a context predictor. Different from last value predictor and stride predictor, a context predictor consists of two tables–a VHT and a Pattern History Table (PHT). For each instruction to predict, the VHT maintains the last history-depth number of unique outcomes produced by the instruction in the "Value History" field. The VHT also maintains a bit field, labeled "Value History Pattern

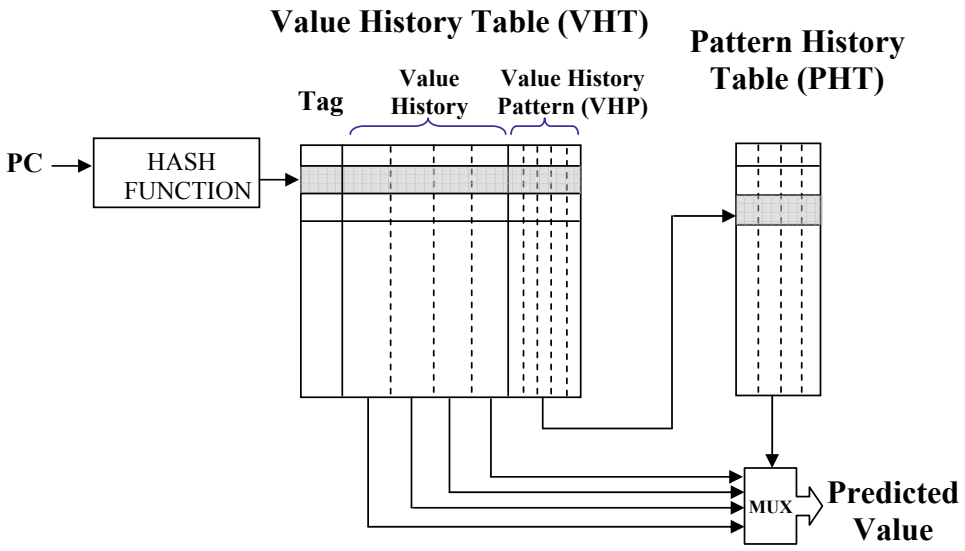

Figure 5.3: Diagram of Last Value Predictor.

(VHP)", that encodes the pattern in which these outcomes occurred during the last pattern-length dynamic instances of the instruction. During prediction, the instruction's VHP field is used to index the PHT. Each PHT entry contains several frequency counters, one for each instruction outcome in the VHT. The counter with the highest count indicates the most frequent successor value given the instruction's current value pattern. If the highest counter is above some threshold, then the corresponding outcome in the "Value History" field is predicted for the instruction; otherwise, no prediction is made. After an instruction has produced its actual computation result, the corresponding PHT entry counter is incremented by an hit bonus while the other counters in the same PHT entry are decremented by a miss penalty. And the corresponding VHP field in the VHT is also updated to reflect the instruction's new outcome pattern.

Hybrid Predictor Hybrid predictor consists of two predictors, usually a stride predictor and a context-based predictor. Prediction from the context predictor has

more preference; if the context predictor cannot make a prediction, then the value predicted–if any–by the stride predictor is selected. Hence, hybrid predictor can perform better since it incorporates the abilities of both the stride and the context predictor–it can predict instruction outputs that have not been seen, but conform to a fixed stride pattern with outputs that have been produced; it can also predict instruction outputs that have appeared, but conform to more complex patterns than simply striding with other precedent output values.

5.1.2 Predictor-Based Fault Detection

To utilize value predictability for fault detection, instruction outputs are first predicted and then compared with actual computational results. In contrast to traditional applications of value prediction which require prediction results at the early stages of the pipeline for breaking true data dependences, to compare with computational results, prediction results are not needed until the writeback stage. This relaxes the timing constraints for the prediction, enabling large and more sophisticated predictors for high prediction rate. Although aggressive predictors are possible, we find simple predictors which incur low hardware overhead and low power consumption can provide significant fault tolerance benefits. In our study, we adopt a hybrid predictor which is composed of one stride predictor and one context predictor. Prediction from the context predictor is selected first; if the context predictor cannot make a prediction, then the stride predictor is consulted to produce a prediction.

During each predictor comparison, the prediction and actual computation result will either match or differ. Each case has two possible interpretations. When the results match, the first possibility is the predictor predicted the correct value. In this case, no fault occurred since the instruction also produced the same correct value. The second possibility is the predictor predicted the wrong value, but a fault occurred such that the instruction produced the same wrong value. This case is highly unlikely, especially under the assumption of Single-Event-Upset (SEU) fault model in our study. Hence, if the prediction and actual computation result match, we assume no fault has occurred, and thus, do not take additional action.

If the prediction and actual instruction result differ, the first possibility is the predictor predicted the correct value. In this case, a fault has occurred since the instruction produced a different value. The second possible interpretation is the predictor predicted the wrong value, and the instruction either produced a correct or wrong value (again, we assume a misprediction and incorrect result will never match). Unfortunately, there is no way to tell which of these cases has occurred, so we can only assume that there is the *potential* for a fault. In our study, we always assume conservatively that a fault has occurred, and initiate recovery by squashing the pipeline and re-executing the squashed instructions in the hopes of correcting the fault. During re-execution, if the instruction produces the same result, then we know with high probability that the original instruction did not incur a fault. If no fault occurred (the most likely case), the pipeline flush was unnecessary, and performance is degraded. (However, as we will see in Section 5.1.3, such "unnecessary" flushes can actually improve reliability in many cases).

To reduce the performance degradation caused by false positives, we use confidence estimation to limit predictions to instructions that have high confidence. In our experiments, we employ the confidence estimator described in [1]. We associate a saturating counter with each entry in the value predictor table. A prediction is made only when the corresponding saturating counter is equal to or above a certain threshold. If the prediction turns out to be correct (the match case), the saturating counter is incremented by some value. If the prediction turns out to be incorrect (the mismatch case in which the original and re-executed results are the same), the saturating counter is decremented by some value. Given confidence estimation, since it is the confidence threshold that determines which instructions to predict ultimately, we can tradeoff the number of false positives with the number of predicted instructions (and hence, the fault coverage) by varying the confidence threshold. Section 5.2 will discuss how we select confidence thresholds.

In addition, we assume both the stride and context predictors used in our work can always make predictions for the selected instructions by their writeback stage. Such assumption is based on the small size of the predictors (see predictor parameters in Section 5.2.1)–our predictors are smaller or equal to the value predictors adopted in other work [1, 35, 36], which have to make predictions by the issue stage for the purpose of boosting performance. Hence there will be no timing-related problems when integrating our predictors into existing CPU pipelines.

5.1.3 Fault Recovery

When an instruction's prediction result differs from its computed value, it is possible that some fault has occurred before or during the instruction's execution. To recover from the fault, it is necessary to roll back the computation prior to the fault, and re-execute. For such purpose, checkpoint is usually used to restore processor state, which unfortunately involves certain hardware or software support as well as performance cost. In our work, we take a simpler approach and perform roll back by flushing from the pipeline the potentially corrupted instructions. After flushing, we re-fetch and re-execute from the flush point. (A similar mechanism for branch misprediction recovery can be used for our technique).

Note our recovery mechanism is not perfect. It can only correct fault corruptions that occur on predicted instructions, or instructions that are downstream from a mispredicted instruction (which would incur flush including the mispredicted instruction, as well as all subsequent instructions). If a fault attacks a non-predicted instruction that is not flushed by earlier mispredicted instructions, then even if the fault propagates to an instruction to be predicted later on, recovery would not roll back the computation early enough to re-execute all the faulty instructions. However, even with this limitation, we find our technique is still quite effective.

Because soft errors are rare, most recoveries are triggered by the mispredictions of the value predictor. As mentioned in Section 5.1.2, such false positives can degrade performance. However, they can also improve reliability. Often times, re-executed instructions run faster than the original instructions that were flushed (the flushed

instructions can prefetch data from memory or train the branch predictor on behalf of the re-executed instructions). As a result, the re-executed instructions occupy the instruction queues for a shorter amount of time, reducing their vulnerability to soft errors compared to the original instructions. This effect is particularly pronounced for instructions that stall for long periods of time due to cache misses. Hence, while false positives due to mispredictions can degrade performance, this degradation often provides a reliability benefit in return. The next section describes how we can best exploit this tradeoff.

5.1.4 Analysis of Instruction Vulnerability

In order to reduce the chance of mispredictions and unnecessary squashes, we not only apply confidence estimation (as described in Section 5.1.2), but also limit value prediction to instructions that are most closely related to overall program reliability–in other words, protecting those instruction can potentially benefit program reliability the most. This section describes how we assess the reliability impact of different instructions.

Recently, to reason about hardware reliability, many computer architects have used Architectural Vulnerability Factor (AVF), which is proposed by Mukherjee *et al* [18]. AVF captures the probability that a transient fault in a processor structure will result in a visible error in program final outputs. It provides a quantitative way to estimate the architectural effect of fault derating. To compute AVF, bits in a structure are classified as being related to architecturally correct execution (ACE

bits), or not (un-ACE bits). Only errors in ACE bits can possibly cause output errors. A hardware structure's AVF is on average the percentage of ACE bits that occupy the hardware structure throughout program execution.

To identify ACE bits and compute AVF, instructions must first be distinguished as ACE or un-ACE. [18] proposed a conservative method–bits (instructions) are ACE unless they can be proved otherwise. The authors identified 5 architectural un-ACE sources including NOP instructions, performance-enhancing instructions (e.g., prefetches), predicated-false instructions, dynamically dead code, and logical masking.[1] Among ACE instructions, we make the key observation that they are not equal in affecting system vulnerability. Instead, each instruction's residence time in hardware structures determines its reliability contribution. As stated by Weaver, et al. [37], the longer time an instruction spends in the pipeline, the more it is exposed to sources of soft errors such as neutron and alpha strikes, and hence, the more susceptible it becomes to faults. To minimize the long residency of ACE instructions, they proposed squashing instructions that incur long delays such as cache misses. We extend this idea by quantifying fault vulnerability at the instruction level, and selectively protecting the instructions that are more susceptible to faults.

To characterize fault vulnerability of each ACE instruction, we measure the fraction of overall AVF it contributes. More specifically, for the hardware structures

[1]Although bits for ACE instructions are not necessarily ACE themselves, (e.g., logical masking may cause some bits of ACE instructions to be un-ACE), we find it is quite related between the statistics of ACE instructions and AVF estimation. In our study, we use the change in the percentage of ACE instructions to reflect the impact on system reliability.

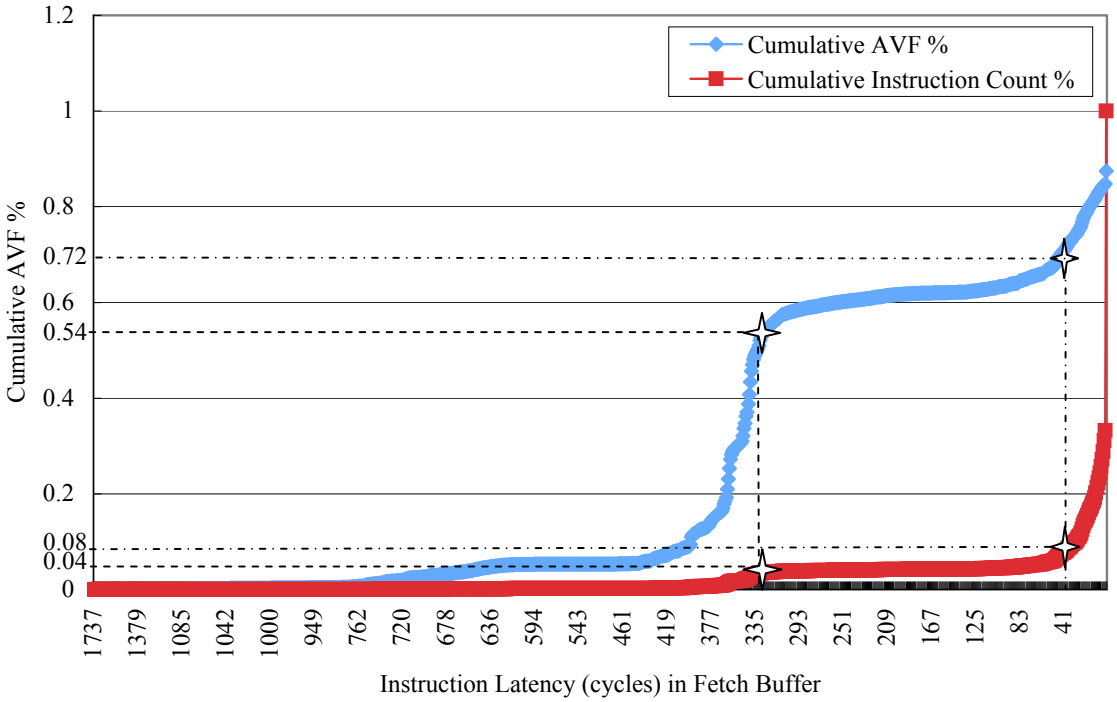

Figure 5.4: Accumulative Percentage of AVF and Instruction Count in Fetch Buffer on TWOLF.

in our study–*e.g.*, fetch buffer and issue queue, we first collect the residence time of each ACE instruction. Next, we sort instructions by their residence time, and compute their corresponding contribution to the structure's total AVF. Then, we plot the cumulative percentage of ACE instructions as well as their cumulative AVF percentage.

Our study shows that a very small number of instructions account for a majority of the AVF in hardware structures. Figure 5.4 illustrates the distribution of ACE instructions in fetch buffer for benchmark TWOLF from SPEC2000 suite. In Figure 5.4, the X-axis represents the accumulated fraction of total instructions. Each point along the X-axis stands for a group of instructions which have the same residence time in fetch buffer; groups of instructions with longer residence time are displayed first. The Y-axis in the graph represents the corresponding accumulated

percentage of AVF or dynamic instructions–the top curve represents AVF while the bottom curve represents instruction count. From the graph we can see at the beginning (leftside of the graph), both the instruction count and their AVF portions are almost zero, indicating that there are few instructions with extremely long residence time. Then, as instructions with shorter latency are counted in, their AVF portion also gets bigger, while the latter increases much faster till the first marked point (on the left of the graph). The marked points (the two points each on the left side of its curve) show that about 3.5% of instructions corresponds to about 53.9% of average AVF for the hardware structure. Afterwards, the increase in AVF portion turns much slower as the number of instructions increases. Therefore, a small set of instructions–3.5% in the example–consists of a large AVF fraction. Those instructions are much more susceptible to faults than the remaining ones –illustrated by the marked points on the right side of the curves, the majority of instructions (about 91.8%) exhibit a latency smaller than 40 cycles, and account for a relatively small portion of the overall AVF (about 28.4%). We find similar behavior occurs for the other benchmarks as well as for the other hardware structures. Such results show that using our value predictor to target a small number of instructions–those with very large latencies–is sufficient to provide the majority of fault protection. (As mentioned in Section 5.1.2, our technique performs checking at instruction's write-back stage. By then the instruction's latency is known. Hence, it can be determined on time whether to check its computation or not.) This is good news since it will minimize the performance impact of mispredictions. In addition, it's up to designers to decide which portions of instructions are susceptible enough to be protected,

which provides desirable flexibility in system design.

In addition, our study also shows instructions exhibit different fault vulnerability at different hardware structures: while an instruction may stall for a long time in one hardware structure, it may not stall for very long in other structures. In other words, a single instruction can contribute differently to different structures' vulnerability. Thus, an interesting question is how to select the smallest group of instructions that will provide the largest benefit to the whole processor's reliability? In our work, we evaluate three different policies for determining instruction's vulnerability: the total residence time of an instruction from fetch to issue, from dispatch to writeback, or from fetch to writeback. We will present our results in Section 5.2.

5.2 Experimental Methodology and Results

In Section 5.1 we discussed our study on characterizing instruction vulnerability. Our results show that a small set of instructions accounts for a big portion of hardware vulnerability. We also qualitatively analyzed the impact of flushing pipeline on value misprediction: flushing degrades performance, but in some cases it may improve program reliability. We consider such tradeoffs in our design, and use insights from both to drive value prediction and confidence estimation–the latter ultimately determines which instructions will get predicted.

In this section, we present our experimental results of applying value prediction to those highly vulnerable instructions for fault protection. Throughout our exper-

iments, we have adopted two analysis techniques to estimate the reliability impacts of our technique. The main method is AVF analysis, which has been discussed in Section 5.1.4. The other method is fault injection, which has been traditionally–and is still widely–used to evaluate system reliability. Compared to AVF computation, the method of fault injection actually introduce faults into a hardware description of a processor or radiation testing on a physical device, thus it makes it possible to observe fault propagation and characterize fault coverage more accurately. However, its nature of statistical sampling requires large amount of trials, which can cause enormous experimental time and resources to be consumed. Contrastingly, although AVF analysis can only provide a lower bound on the reliability of a processor design, it performs much faster–it utilizes high-level performance model coupled with low-level design information, and only needs one simulation run to obtain the reliability estimate of the design. Since both analysis techniques have their own advantages and disadvantages, we mainly employ AVF analysis to evaluate the reliability impact of our technique (Section 5.2.2, Section 5.2.3, and Section 5.2.4). We also conduct another group of experiments with fault injection (Section 5.2.5), and compare the results from AVF computation and fault injection experiments(Section 5.2.6).

In more detail, Section 5.2.1 first introduces the processor model and the benchmarks we use in our experiments. Then, in Section 5.2.2, we investigate the potential of using value prediction to check the computations of highly vulnerable instructions. We compare our approach with predicting all the result-producing instructions. As we have discussed, with value prediction technique, we exploit program's inherent redundancy for result comparison and fault detection. In addition,

Hybrid Value Predictor Parameters	
VHT size / values per VHT entry	1024 / 4
PHT size / PHT counter threshold	1024 / 3
Confidence Estimator Parameters	
saturation threshold	15
low / mid / high threshold	3 / 7 / 15
miss penalty / increment bonus	7 / 1

Table 5.1: Parameter settings for the detailed architectural model into which we inject faults.

by identifying and protecting the most vulnerable instructions, we can potentially limit performance degradation while maintaining comparable reliability gain. Next, in Section 5.2.3, we incorporate the finding of various vulnerability across different instructions, and propose adaptive threshold in confidence estimation. In Section 5.2.4, we also discuss different policies for computing instruction latency, which is used in selection for value prediction. At last, in Section 5.2.5, we employ fault injection for another group of selective value prediction experiments. Section 5.2.6 presents the comparison between fault injection and AVF computation.

5.2.1 Simulator and Benchmarks

In our experiments, on top of the baseline detailed architectural simulator shown in Table 3.1, we implement a hybrid value predictor including one stride predictor and one context predictor–as described in [35]. We also implement a confidence estimator as described in [1]. Both the value predictor and confidence estimator are configured as specified by Table 5.1.

In addition, Table 5.2 lists all the benchmarks used in our experiments. These

Benchmark	Input	Instruction Count
300.twolf	ref	109546670
176.gcc	166.i	240000000
254.gap	train.in	411061781
164.gzip	input.compressed	192015257
256.bzip2	input.compressed	2346534735
253.perlbmk	diffmail.pl	1000000000
197.parser	ref.in	1404572471
181.mcf	inp.in	500000000
175.vpr	test	1512992144

Table 5.2: Benchmarks and input datasets used in our experiments.

benchmarks come from the SPEC2000 Integer suite. In the table, the column labeled "Input" specifies the input dataset used for each benchmark, and the column labeled "Instruction Count" reports the number of instructions executed by each benchmark–we start simulation after program initialization, so "Instruction Count" does not include the benchmarks' initialization part.

5.2.2 Value Prediction Experiments

We first present our experiments on applying value prediction without confidence estimation to fault detection. We evaluate the impacts on both reliability and performance when predicting all or a portion of result-producing instructions, which we call full or selective prediction, respectively. Full prediction predicts an instruction as long as the predictor can make a prediction (as described in Section 5.1.1, the predictors are unable to make predictions in some cases). Selective prediction only predicts instructions that meet some minimum latency threshold–we measure instruction's latency from the fetch stage to the issue stage.

Figure 5.5 reports the percentage of result-producing instruction count (to the

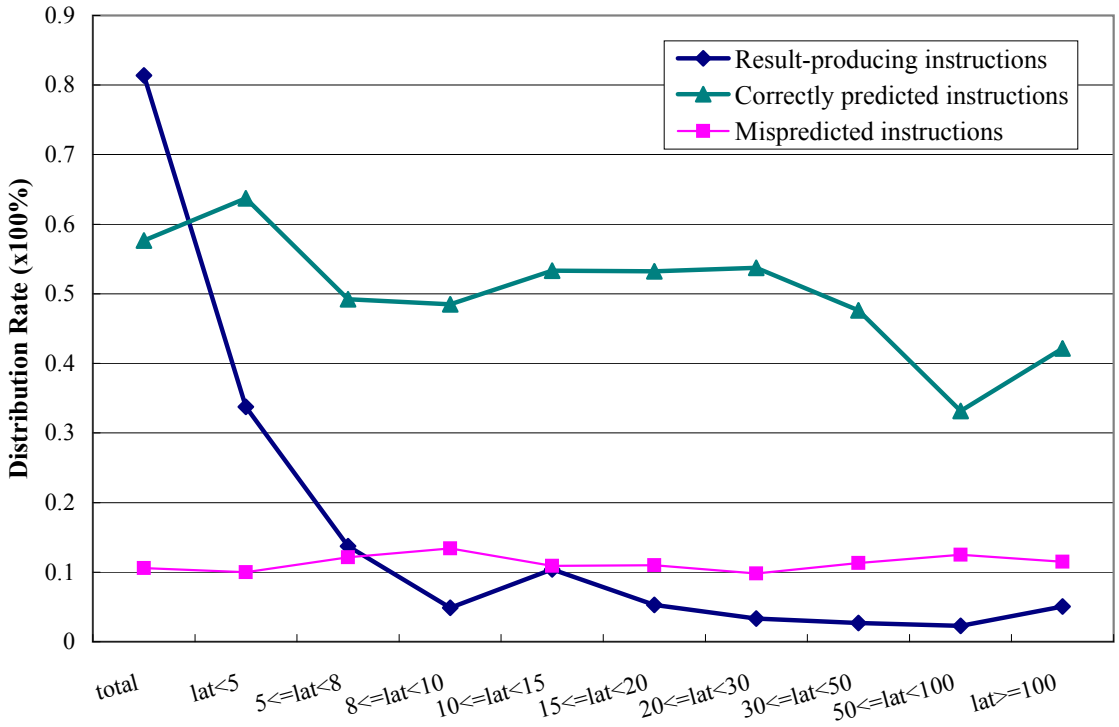

Figure 5.5: Average distribution of result-producing instruction count, as well as their prediction and misprediction rate, across different latency range over all the 9 spec2000 integer benchmarks. Latency is measured from fetch to issue stage.

total number of executed instructions) across all instructions (labeled "total") and for different latency ranges, as well as the fraction of instructions from each category ("total" and the different latency ranges) that are correctly predicted and mispredicted. Every datapoint in Figure 5.5 represents an average across all the benchmarks listed in Table 5.2. In the graph, the X-axis represents instruction latency range, marked as increasing latency while the leftmost stands for the complete set of result-producing instructions; the Y-axis represents either percentage of instruction count, or the percentage of correctly predicted or mispredicted instructions. From the graph, we see in total result-producing instructions account for about 81.4% of all instructions. In particular, instructions with shorter latency–*i.e.*, latency less than 5 cycles from fetch to issue stage–account for about 32.1% of all the instructions, or

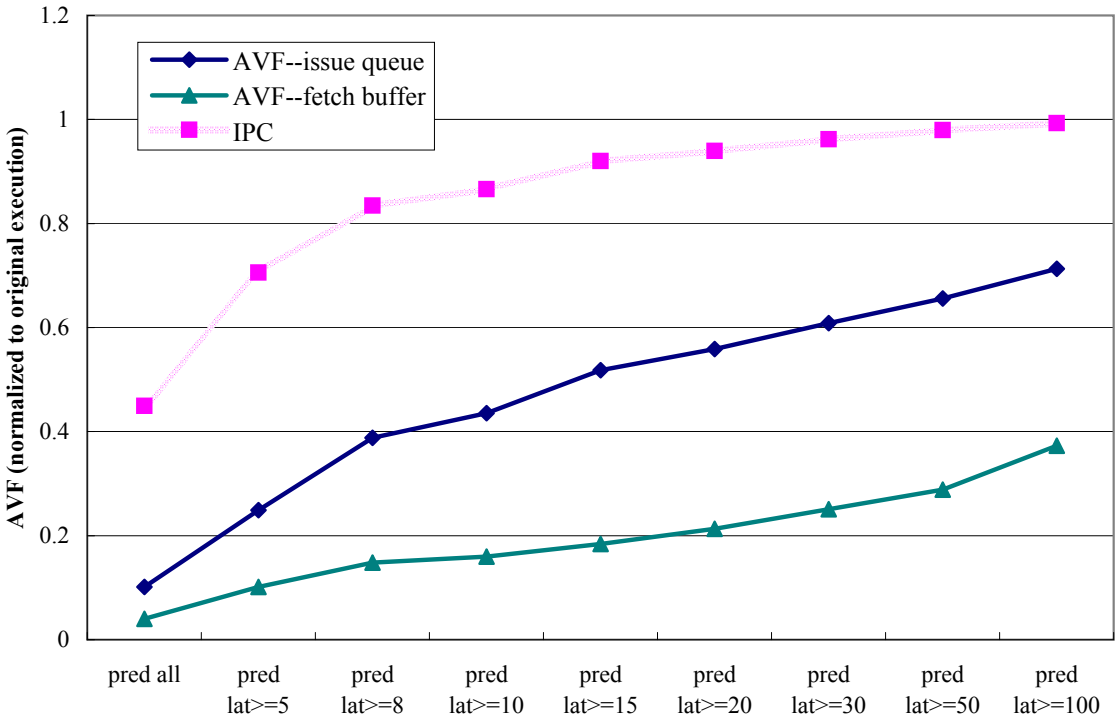

Figure 5.6: Average AVF of 2 hardware structures and IPC (relative to original execution) across 9 spec2000 integer benchmarks by applying value prediction to long-latency instructions. Latency is measured from fetch to issue stage. On value misprediction, mispredicted instructions and all subsequent ones are flushed and then re-fetch and re-execute.

41.4% of result-producing instructions. Moreover, these short-latency instructions exhibit relatively good prediction rates–63.7% on average. In contrast, instructions with longer latency (than 5 cycles) have slightly lower prediction rate–around 40% to 50%. However, given the importance of these long-latency instructions to system reliability, it is still worth to check their values through prediction.

We now measure the impact of value prediction on both program reliability and performance. As we have discussed in Section 5.1.2, on each value misprediction, the mispredicted instruction and all subsequent instructions are flushed from the pipeline, and the program starts to re-fetch and re-execute from the flushed point. Figure 5.6 reports the average IPC and AVF of fetch buffer and issue queue, config-

ured as Table 3.1–by full or selective prediction. Each datapoint in the graph represents the average across all the benchmarks; the X-axis represents latency threshold used for selective value prediction–only instructions that stay in the pipeline (measured from fetch to issue stage) longer than or equal to the corresponding threshold are selected for value prediction; the Y-axis represents the AVF or IPC normalized to the baseline values without prediction. The leftmost points in the graph (marked as "pred all" on the X-axis) represent results from full prediction.

Figure 5.6 shows prediction-based fault protection can be very effective at improving reliability (*i.e.*, reducing AVF). The AVF for the fetch queue and issue queue is reduced by as much as 96.0% and 89.8%, respectively (under full prediction) compared to no prediction. This is due to both correct and incorrect predictions. On a correct prediction, the value of the predicted instruction is checked, so the instruction is no longer vulnerable, and hence, does not contribute to the AVF of the structures it occupies. On a misprediction, the pipeline is flushed. As discussed in Section 5.1.2, re-execution after flushing is typically faster than the original execution, thus reducing the occupancy of ACE instructions in the hardware structures. Both combine to provide the AVF improvements shown in Figure 5.6.

Unfortunately, these reliability improvements come at the expense of performance. Figure 5.6 shows IPC can degrade significantly due to the penalty incurred by mispredictions, particularly when a large number of instructions are predicted. Under full prediction, IPC reduces by 55.1% compared to no prediction. But the performance impact lessens as fewer instructions are predicted (moving towards the right side of Figure 5.6). For example, when predicting instructions with latency

greater than or equal to 5 cycles, the performance impact is less than 29.4%. Although reliability improvement is not as great when predicting fewer instructions, it can still be significant–we achieve a 84.0% and 56.5% reduction in AVF for the fetch buffer and issue queue respectively at threshold latency of 5 cycles. Furthermore, when predicting even fewer instructions, although both the performance and reliability impacts become smaller, they do not reduce at the same pace. For example, when predicting instructions with latency greater than or equal to 30 cycles, the performance impact is less than 3.8%, while the AVF reduction for the fetch buffer and issue queue can still achieve up to 74.9% and 39.2%, respectively.

In general, Figure 5.6 indicates there exists a tradeoff between reliability and performance. The more instructions we predict, the larger the improvement in reliability, but also the larger the degradation in performance. However, when we focus the value predictor on long-latency instructions such as instructions with $\geq$ 30-cycle latency, the performance loss is small while the reliability gain is still quite large. This is because the longer the instruction latency, the smaller the impact mispredictions will have on performance. Furthermore, the longer the instruction latency, the more critical the instructions are from a reliability standpoint.

5.2.3 Confidence Estimation

As value misprediction causes pipeline squash which degrades program performance, we integrate a separate confidence estimator with the value predictor to reduce the number of mispredictions. Figure 5.7 reports the prediction and mispre-

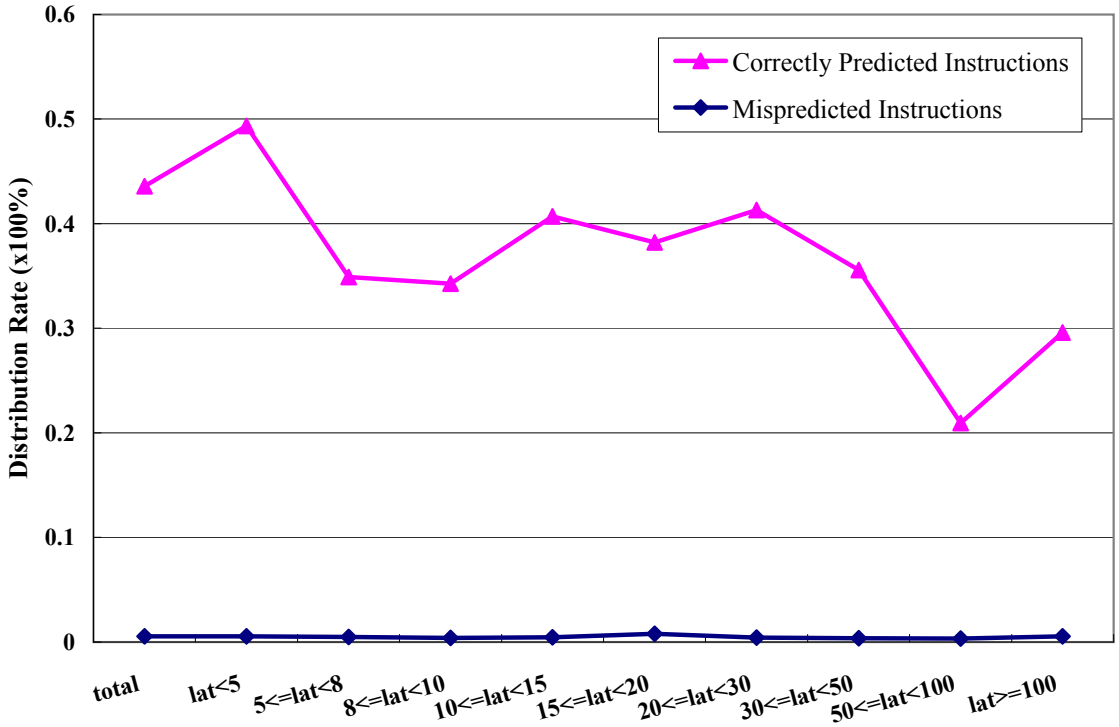

Figure 5.7: Average distribution of value prediction rate–with confidence estimation–across different latency range over all the 9 spec2000 integer benchmarks. Latency is measured from fetch to issue stage.

diction rates of our value predictor with confidence estimation for all instructions, labeled "total", and for instructions with different latency ranges (The graph is presented in the same way as Figure 5.5). The confidence estimator is configured as Table 5.1. Compared with Figure 5.5, which has no confidence estimation applied, Figure 5.7 shows our value predictor achieves fewer correct predictions–the reduction ranges between 10% and 15%. This is because the confidence estimator prevents predicting the less predictable instructions. As a result, the misprediction rate goes down to almost 0 across all latency ranges. As Figure 5.7 shows, our confidence estimator is quite effective at reducing mispredictions with only a modest dip in the number of correct predictions.

Figure 5.8 shows the impacts of confidence estimation on the AVF of our

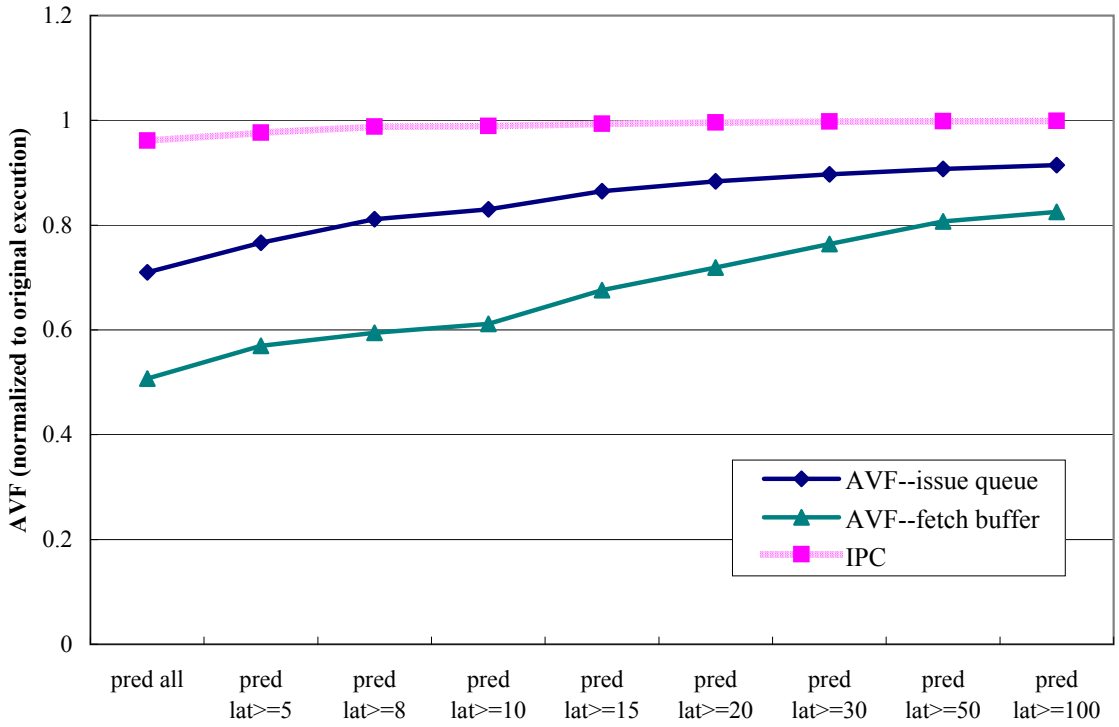

Figure 5.8: Average AVF of 2 hardware structures and IPC (relative to original execution) across 9 spec2000 integer benchmarks by applying value prediction to long-latency instructions. Value predictor is complemented with a separate confidence estimator. Latency is measured from fetch to issue stage. On value misprediction, mispredicted instructions and all subsequent ones are flushed and then re-fetch and re-execute.

hardware structures, as well as on IPC (the graph uses exactly the same format as Figure 5.6). In Figure 5.8, we see IPC never degrades more than 4%, even when performing full prediction. These results show confidence estimation is indeed effective at mitigating performance degradation. Unfortunately, applying confidence estimation also brings down the reliability improvement. In particular, under full prediction, the AVF for the fetch buffer and issue queue is now reduced by at most 49.3% and 29.0%, respectively; under selective prediction, the AVF for the fetch buffer and issue queue is reduced by about 43.1% and 23.4% for prediction with baseline latency of 5 cycles, and 23.6% and 10.3% for prediction with base-

line latency of 30 cycles, respectively. Such lower reliability improvements are still exposed after taking performance loss into account. In all, by comparing reliability and performance impacts between with confidence estimation (Figure 5.8) and without confidence estimation (Figure 5.6), we see confidence estimation is helpful in lessening performance loss due to mispredictions, but it also degrades reliability improvement since it suppresses prediction of many instructions, thus reduces the coverage achieved by the value predictor.

Thus far, we have applied confidence estimation uniformly across all eligible instructions–*i.e.*, we use a single confidence threshold to determine whether a particular instruction should be predicted or not. However, predicting all instructions using a uniform confidence level may not be the best policy since instructions do not contribute equally to program reliability. In particular, for longer latency instructions which are more susceptible to faults and thus contribute more to overall reliability–they usually also incur less performance degradation during misprediction recovery, it may be better to perform value prediction more aggressively. Conversely, for shorter latency instructions which contribute less to overall reliability–they usually incur more performance degradation during recovery compared to instructions with longer latencies, it may be better to perform value prediction less aggressively. This suggests an adaptive confidence estimation technique has the potential to more effectively tradeoff reliability and performance.

In our study, we modify our confidence estimation scheme to adapt the confidence threshold based on each instruction's latency. In particular, we employ three different threshold levels, similar to what is proposed in [1]. (The thresholds for low,

medium, and high confidence are 3, 7, and 15, respectively for a saturating value of 15). Instructions which latencies falls in certain ranges are assigned corresponding confidence thresholds: we use the lowest confidence threshold for instructions that incur a latency equal to or larger than 4 times the baseline latency; we use the medium confidence threshold for instructions that incur a latency equal to or larger than 2 times the baseline latency but smaller than 4 times the baseline latency; and we use the highest confidence threshold for instructions that incur a latency equal to or larger than the baseline latency but smaller than 2 times the baseline latency. Here, the baseline latency is the minimum instruction latency that is considered for prediction as given by latency-based selective prediction. (For example, if we only predict instructions with latency 5 cycles or larger, then the low, medium, and high thresholds are applied to instructions with latency in the ranges $\geq$ 20 cycles, 10-19 cycles, and 5-9 cycles, respectively).

We measure both reliability and performance impacts of combining latency-based selective prediction with adaptive confidence estimation as described. Figure 5.9 reports the new AVF of our three hardware structures, as well as IPC, averaged across all the benchmarks. The graph is plotted in the same format as Figure 5.6 and Figure 5.8, except that now, each latency marked along the X-axis in the graph represents the smallest (baseline) latency threshold–only instructions that stay longer than the baseline latency are eligible for prediction, which is also associated with the highest confidence threshold; the corresponding medium and largest latency thresholds are by default two or four times of the baseline latency, and are associated with the medium or lowest confidence threshold, respectively. Only

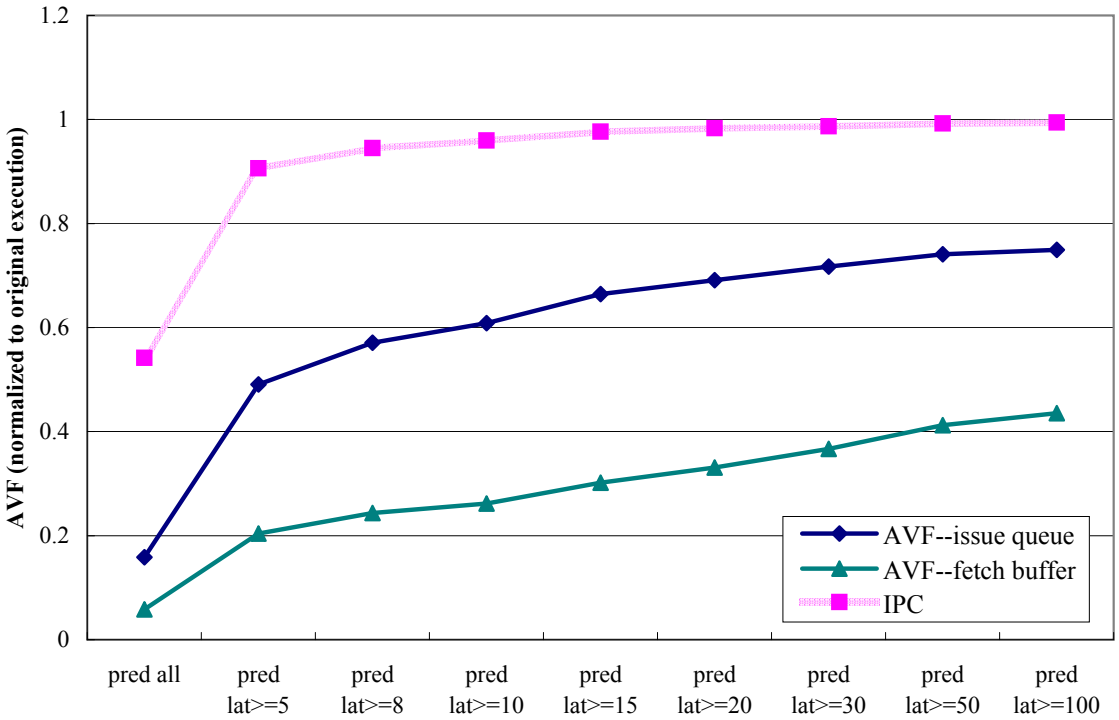

Figure 5.9: Average AVF of 2 hardware structures and IPC (relative to original execution) across 9 spec2000 integer benchmarks by applying value prediction to long-latency instructions. Latency is measured from fetch to issue stage. Confidence threshold used for each prediction varies (high, medium or low threshold) according to the instruction's latency. On value misprediction, mispredicted instructions and all subsequent ones are flushed and then re-fetch and re-execute.

instructions whose saturating counters meet the corresponding confidence threshold are predicted. Comparing with the baseline confidence estimation technique shown in Figure 5.8, Figure 5.9 shows that similarly, adaptive confidence estimation incurs a relatively small performance degradation. For example, under selective prediction, the performance degradation is about 9.4% for prediction with 5-cycle baseline latency, and 2.4% for prediction with 15-cycle baseline latency. However, adaptive confidence estimation achieves a much better reliability improvement (AVF reduction) than the baseline confidence estimation, and approaches the reliability improvement achieved by value prediction without confidence estimation shown in

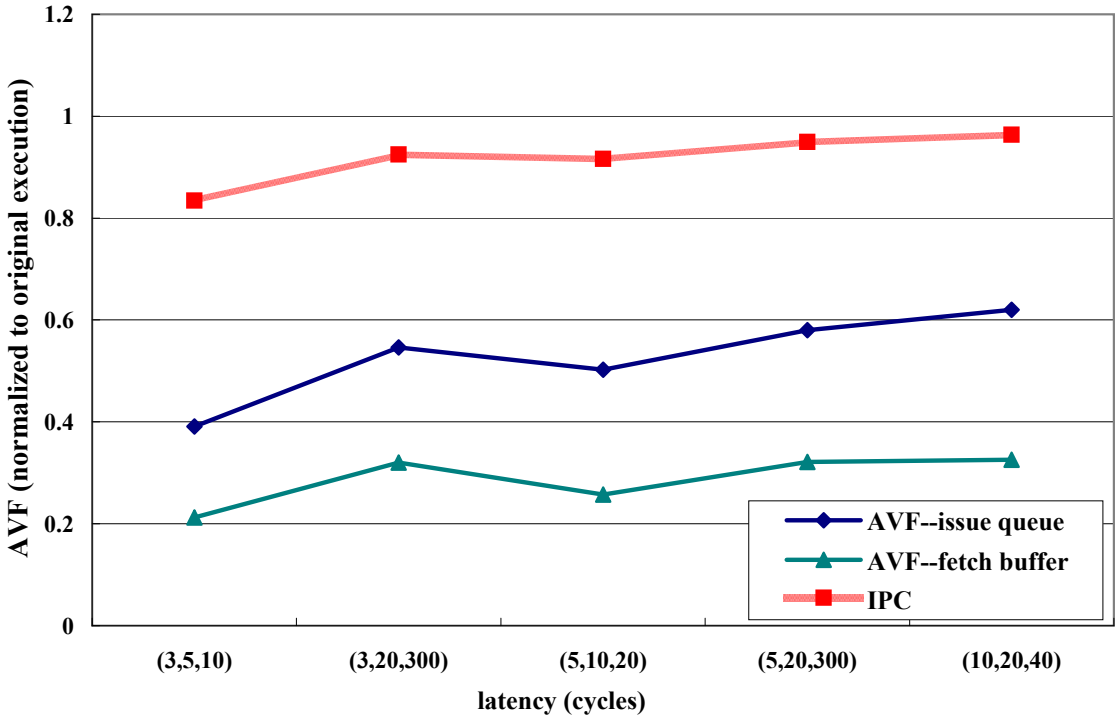

Figure 5.10: Average IPC and AVF of 2 hardware structures across 9 spec2000 integer benchmarks by varying threshold of confidence estimation according to instruction latency. Each set of values marked along X-axis contains three latency thresholds, listed as lowest, medium, and highest threshold, and associated with three confidence thresholds, respectively– the three confidence thresholds are selected as described in [1].

Figure 5.6. For example, under selective prediction, the AVF for the fetch buffer and issue queue is reduced by 79.6% and 50.9% for prediction with baseline latency of 5 cycles, and 69.8% and 33.6% for prediction with baseline latency of 15 cycles, respectively. Thus, by more aggressively predicting only the longer latency instructions, adaptive confidence estimation can cover the most critical instructions for reliability without sacrificing too much on performance.

In addition to our 3-level confidence-threshold policy, we have also tried other combinations of instruction latencies with confidence thresholds, which impacts on reliability are exhibited in Figure 5.10. In Figure 5.10, the X-axis represents 5 sets of

instruction latency threshold, while each set contains three latency thresholds, listed as lowest, medium, and highest latency threshold. The three latency thresholds in each set are associated with the three confidence thresholds as described above. The association between latency and confidence threshold is: instructions which latencies are equal to or longer than the highest latency threshold are assigned with the lowest confidence threshold; instructions which latencies are shorter than the highest latency threshold, but equal to or longer than the medium latency threshold are assigned with the medium confidence threshold; while instructions which latencies are shorter than the medium latency threshold, but equal to or longer than the lowest latency threshold are assigned with the highest confidence threshold. Note the two leftmost sets of latency threshold in the graph–(3, 5, 10) and (3, 20, 300)–belong to full prediction since in our processor model, the smallest latency for an instruction from fetch to issue in our processor model is 3 cycles. The other three sets of latency threshold along the X-axis are for selective prediction. In particular, the middle set, (5, 10, 20), satisfies our 3-level confidence-threshold policy as described before. In addition, the Y-axis in Figure 5.10 stands for either IPC or AVF on one hardware structure; each point in the graph stands for the average over experiments on all the benchmarks in our study. Figure 5.10 indicates that full prediction (such as the leftmost points in the graph) with varying confidence threshold has less advantage on reliability gain when considering performance (IPC); on the contrary, selective prediction–such as the datapoints in the middle of the graph with latency range (5, 10, 20)–results in less IPC reduction with good AVF gain. We adopt the set of (5, 10, 20) as the latency thresholds for the experiments in Section 5.3.

5.2.4 Policy on Measuring Instruction Latency for Value Prediction

In the experiments presented so far, we measure each instruction's residence time from fetch to issue stage to determine its criticality for program's fault vulnerability. This can be implemented with a unified tag to keep the total processing time of an instruction including fetch, dispatch, and issue. However, as value prediction occurs at the end of execution stage (or the start of writeback), there exist other choices for such criticality measurement. In our study, we compare three policies–calculating instruction's latency from fetch to issue, from dispatch to writeback, or from fetch to writeback–for the resulting reliability and performance impacts.

We report reliability and performance impacts of using the three kinds of processing time on our hardware structures: we use Figure 5.11 to show the new AVF results and Figure 5.12 to show the new IPC results. In both graphs, policy 1 stands for latency computed from fetch to issue stage, policy 2 is for latency from dispatch to writeback, and policy 3 is for latency from fetch to writeback stage. The X-axis in the two graphs represents the same meaning as in Figure 5.6, Figure 5.8, or Figure 5.9. From the graphs we can see, generally, there is no big difference on issue queue among using different policy for computing instruction latency, while for fetch buffer, incorporating fetch time–*i.e.*, from fetch to issue or fetch to writeback stage–ends up with more reliability gain than using dispatch to writeback time. Such results are reasonable since fetch time directly indicates an instruction's vulnerability at fetch buffer, while it is counted in policy 1 and 3 but ignored in policy 2. We stick to policy 1–counting instruction latency from fetch to

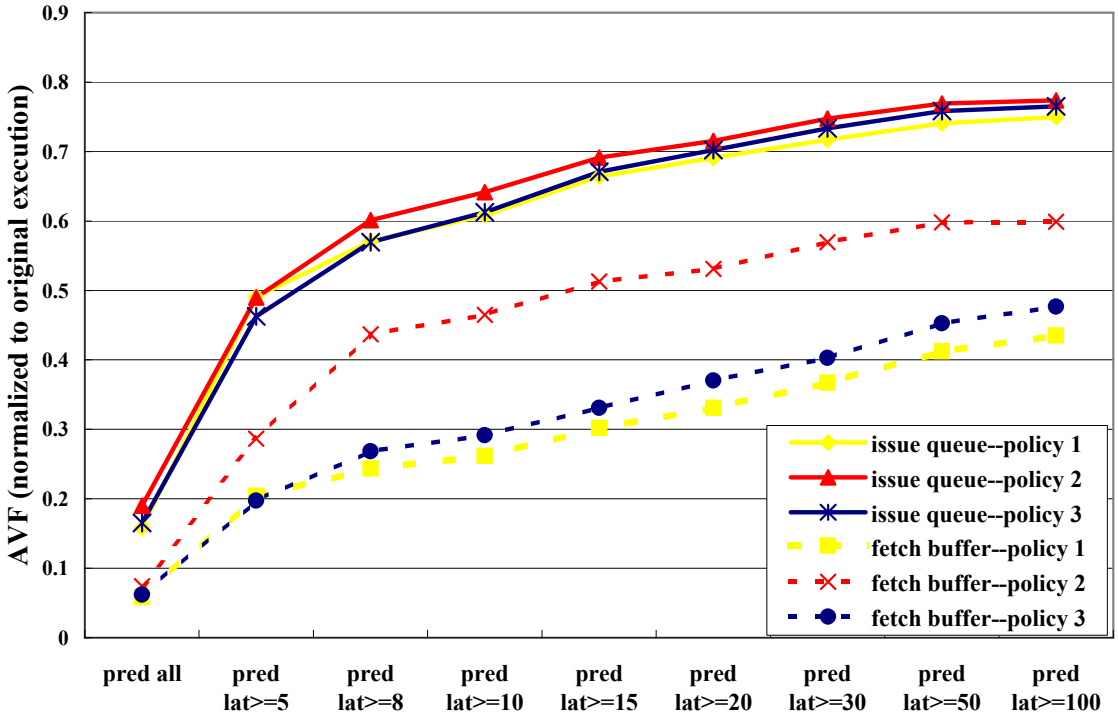

Figure 5.11: Average AVF comparison of 3 policies for computing instruction latency–instructions are selected by their latency for value prediction and confidence estimation–on fetch buffer and issue queue (relative to original execution) across 9 spec2000 integer benchmarks. Policy 1 stands for latency computation from fetch to issue stage, policy 2 is for latency from dispatch to writeback, and policy 3 is for latency from fetch to writeback stage. The confidence threshold varies with instruction latency. On value misprediction, mispredicted instructions and all subsequent ones are flushed and then re-fetch and re-execute.

issue stage–in Section 5.3.

5.2.5 Selective Value Prediction Experiments with Fault Injection

Thus far, we have investigated the reliability impact of our technique through AVF analysis. As we have discussed, through AVF analysis, we can estimate the system reliability very quickly–only one simulation round is needed. However, AVF analysis is conservative and it can only provide a lower bound on the reliability of the system design. Such conservatism mainly comes from a few sources. First,

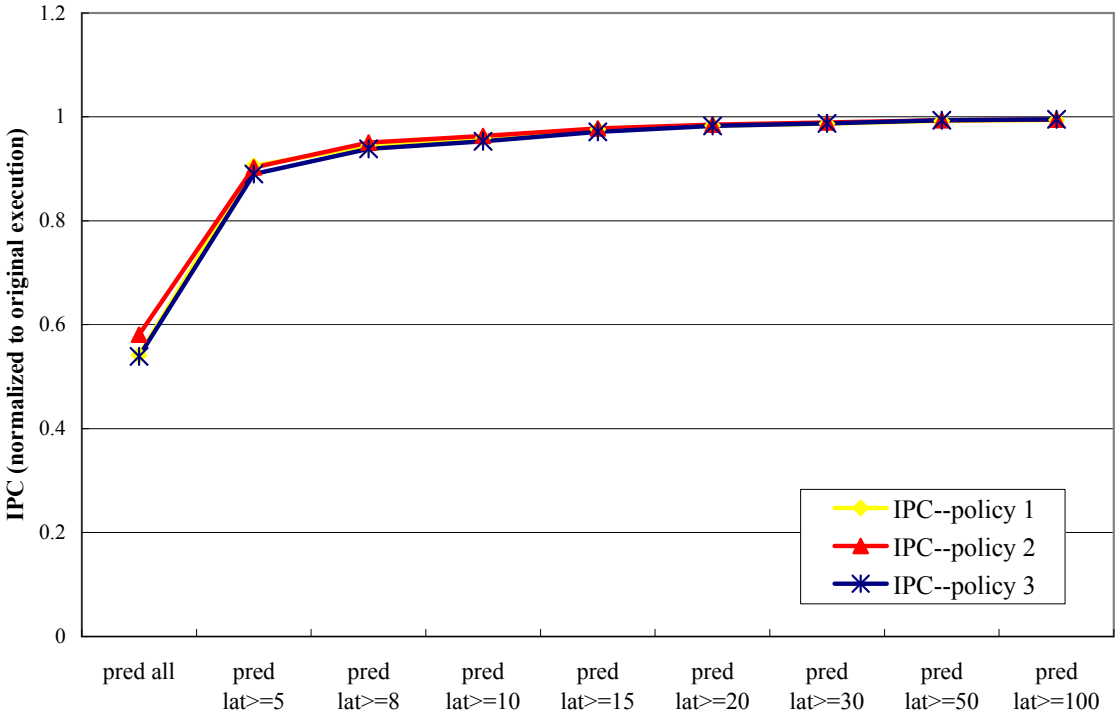

Figure 5.12: Average IPC comparison of 3 policies for computing instruction latency–instructions are selected by their latency for value prediction and confidence estimation–on fetch buffer and issue queue (relative to original execution) across 9 spec2000 integer benchmarks. Policy 1 stands for latency computation from fetch to issue stage, policy 2 is for latency from dispatch to writeback, and policy 3 is for latency from fetch to writeback stage. The confidence threshold varies with instruction latency. On value misprediction, mispredicted instructions and all subsequent ones are flushed and then re-fetch and re-execute.

AVF analysis is typically conducted on a high-level performance timing model, thus lacks for enough detail of the system design. Second, a system's AVF is computed by first identifying un-ACE bits as many as possible and then assuming all the remaining bits are ACE. Thus, un-ACE bits that are not identified are all counted as ACE, which causes the computed AVF result to be bigger than its actual value. In addition, in analyzing the reliability impact of our technique, we count predicted instructions as un-ACE (in addition to the un-ACE bits identified in [18]). This is conservative because with value prediction, not only instructions that can be correctly predicted are covered from faults, but also instructions that feed values

to them may also have executed normally. For example, as we have mentioned, for arithmetic instructions, their input data has to be unique if all other inputs and outputs have certain values (correct prediction). Thus, instructions that have produced the corresponding input values may also have executed properly, which means the state bits that have contributed to the related computation of those instructions are un-ACE, too. Omitting those un-ACE bits results in higher AVF than the actual vulnerability of the system design.

To verify and compare with the AVF results, we conduct another group of experiments through fault injection. The fault injection experiments are performed on the same detailed architectural simulator that models a modern out-of-order superscalar as described in Section 5.2.1. The simulator settings are also the same as listed in Table 3.1. We set our value predictor so that it selectively predicts instructions that stay equal to or more than 5 cycles in pipeline from fetch to issue stage. As for confidence estimation, we adopt the set of latency threshold (5, 10, 20) to associate with the highest, medium, and lowest confidence threshold, separately (related analysis is discussed in Section 5.2.3). Since we choose the same configuration for both the simulator and the value predictor, the performance impact is the same as reported in Figure 5.9–*i.e.*, the average IPC across all the benchmarks listed in Table 5.2 is degraded by 9.4% relative to excution without value prediction.

To estimate the reliability impact, we inject faults into three hardware structures: the physical register file, the fetch queue, and the issue queue (IQ). Similar to our previous work, we assume faults injected into a physical register will appear in architectural state unless the register is idle or belongs to a mispeculated instruction.

Benchmark	Exec Time	Inter Time	Injects	Regfile	Fetch	Issue
300.twolf	138292502	50000.0	2782	905 (0.32)	709 (0.25)	411 (0.15)
176.gcc	169540504	50000.0	3367	149 (0.05)	2327 (0.32)	210 (0.06)
254.gap	248111790	50000.0	8491	1391 (0.28)	652 (0.13)	480 (0.10)
164.gzip	93443879	50000.0	1899	441 (0.24)	279 (0.15)	284 (0.15)
256.bzip2	732651712	250000.0	2941	1132 (0.38)	1650 (0.56)	773 (0.26)
253.perlbmk	635694346	250000.0	2466	1272 (0.49)	782 (0.32)	393 (0.16)
197.parser	1065840259	250000.0	4301	1060 (0.25)	811 (0.19)	750 (0.18)
181.mcf	3733522703	250000.0	14877	4366 (0.29)	10363 (0.70)	5229 (0.35)
175.vpr	807673917	250000.0	3248	1016 (0.31)	202 (0.06)	388 (0.12)

Table 5.3: Detailed fault injection statistics for benchmarks used in our study. "Exec Time" reports the execution time in cycles for each benchmark without value prediction(original run). "Inter Time" reports the average time (cycles) between fault injections. "Injects" reports for original run, the total number of faults injected into the physical register file. The last 3 columns report the number of faults that fall on non-speculative instructions or related resources for the physical register file, fetch queue, and issue queue, respectively.

For the fetch queue, we allow faults to corrupt instruction bits (including opcodes, register addresses, and immediate specifiers) and instruction address (PC). Faults on PC for branch instruction may corrupt its computation. These faults manifest in architectural state as long as the injected instruction is not mispeculated. Lastly, for the IQ, we model 6 fields per entry: instruction opcode, 3 register tags (2 source and 1 destination), an immediate specifier, and a PC value. Like the fetch queue, faults in the IQ appear in architectural state for instructions that are not mispeculated. Corruptions to the IQ opcode and immediate fields behave similarly to those in the fetch queue. Corruptions to the register tags alter instruction dependences, and corruptions to the PC value affect branch target addresses.

We perform fault injections across all the 9 SPEC2000 integer benchmarks in our study, then compare reliability results of their original runs with those implemented with value predictor. Table 5.3 presents fault injection information for

the base-case runs of each benchmark. First, the column labeled "Exec Time" reports for each benchmark's base case, their measured execution time in cycles for running a number of instructions (listed in Table 5.2) on our detailed out-of-order simulator. In our experiments, we inject faults only after program initialization, so "Exec Time" does not include the benchmarks' initialization phase. After program initialization, we perform fault injections on a single hardware structure. We perform 3 such injection runs on each benchmark to inject faults into the 3 hardware structures (physical register file, fetch queue, and issue queue). During each run, faults are randomly injected into a hardware structure one after another using a uniformly distributed inter-fault arrival time. The column labeled "Inter Time" in Table 5.3 reports the inter-fault arrival time (in cycles) used for each benchmark, while the column labeled "Injects" reports the total number of injected faults for the physical register file. (The number of injected faults for the other two hardware structures is almost identical since they use the same inter-fault arrival time). For program execution implemented with value prediction, their execution time varies from executing the same instructions in their original runs due to flushes on misprediction, which also causes different number of faults to be injected since we keep the same inter-fault time for each benchmark.

Among all the faults injected, faults that attack idle hardware resources or hardware occupied by mispeculated instructions have no impact on program execution– they have been masked by the microarchitecture. We ignore these faults and only simulate the remaining ones that somehow disrupt program control or data flow computation. We continue simulation until faults are removed or the originally cor-

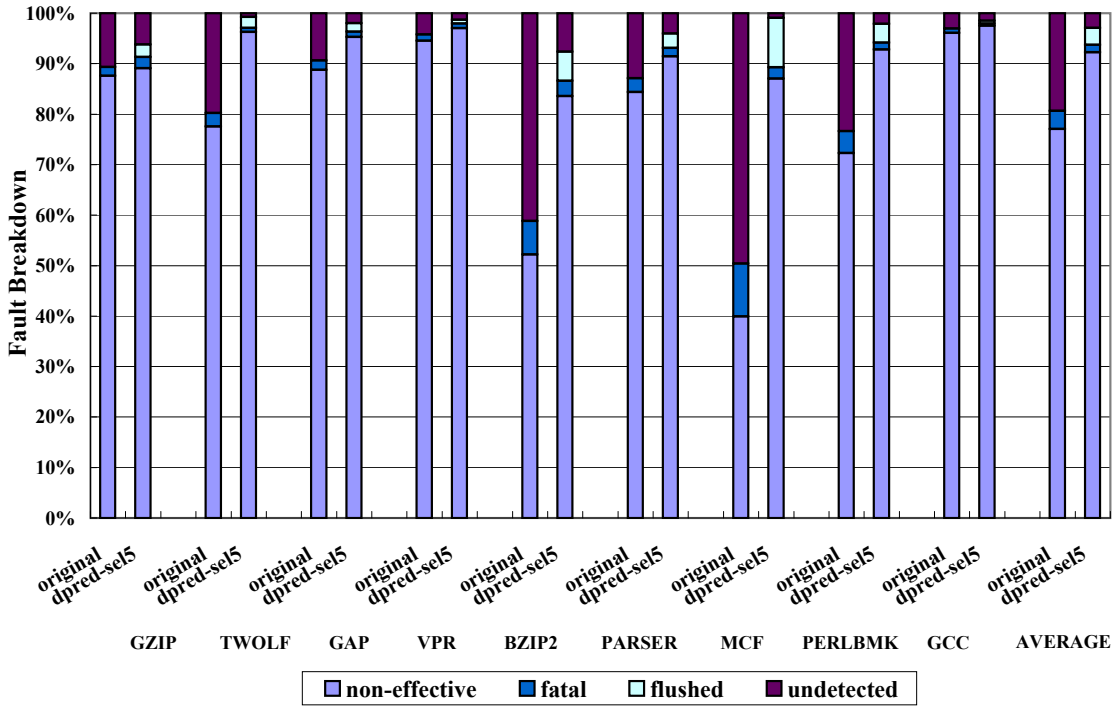

Figure 5.13: Breakdown of fault injections on fetch buffer by applying our selective value prediction technique. The minimum latency threshold for prediction is 5 cycles, measured from fetch to issue stage. Confidence threshold used for each prediction varies (high, medium or low threshold) according to the instruction's latency. On value misprediction, all the instructions in the pipeline are flushed and then re-fetch and re-execute. Categories include faults that have no architectural impact ("non-effective"), faults that cause program to crash or deadlock ('fatal"), faults that are removed during pipeline flushes before faulty instructions commit ("flushed"), and faults that are not detected by value predictor before faulty instructions commit ("undetected").

rupted instructions commit (for fault injection on physical register, we also check if the latest instruction which updates the register value has committed). We define the latter condition as "failure". Such definition excludes fault masking effects from software itself, which usually requires tracking until the end of program execution. Instead, we focuses on faults' architectural impact which examines the sensitivity of value predictor to faults.

Figure 5.13 breakdowns all the fault injections on fetch buffer by their effects

on program execution. In Figure 5.13, each group of bars represents experiments on one benchmark, while the last group represents the average across all the benchmarks. In each group, the bar labeled "original" represents the original execution, while the bar labeled "dpred-sel5" represents the execution implemented with our selective value prediction technique. Each bar contains four categories. The first category, labeled "non-effective", represents the portion of faults which cause no architectural change compared to fault-free execution. For example, faults fall on entry bits that are idle or contain speculative instructions, or the instructions are non-speculative but the bits do not affect the computation, hence the faults are also masked. The second category is labeled "fatal" and includes faults that cause program to crash or deadlock. Then, the category labeled "flushed" represents faults that are removed during pipeline flushes before faulty instructions commit. Lastly, the category labeled "undetected" corresponds to faults that are not detected by value predictor before faulty instructions commit. From the graph, we see the category of "non-effective" consists of the majority of the fault injections–for original program execution, on average about 77.0% of all the faults have no effect on program execution, while for program execution with value prediction, about 92.2% of all the faults have no effect. The higher rate of "non-effective" category is mainly due to fault recovery in the value prediction technique–flushing the pipeline causes more bits in the hardware to sit idle which therefore appear invulnerable to faults. In addition to the faults that do not affect program execution, about 3.6% and 1.5% of all the faults for program execution with and without value prediction, respectively cause program crashes or deadlock before the corrupted instructions commit. These

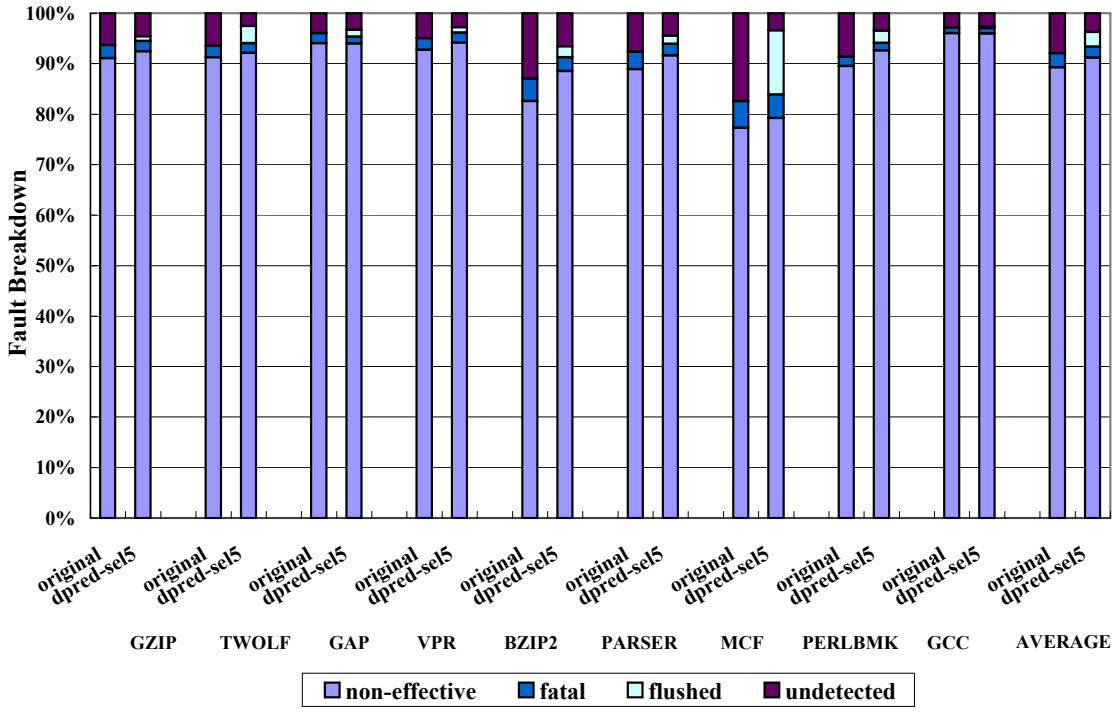

Figure 5.14: Breakdown of fault injections on issue queue by applying our selective value prediction technique. The minimum latency threshold for prediction is 5 cycles, measured from fetch to issue stage. Confidence threshold used for each prediction varies (high, medium or low threshold) according to the instruction's latency. On value misprediction, all the instructions in the pipeline are flushed and then re-fetch and re-execute. Categories include faults that have no architectural impact ("non-effective"), faults that cause program to crash or deadlock ('fatal"), faults that are removed during pipeline flushes before faulty instructions commit ("flushed"), and faults that are not detected by value predictor before faulty instructions commit ("undetected").

faults can be detected by the system automatically. The remaining faults, about 19.3% for original program execution corrupt the computation of some instructions which still manage to commit. Contrastingly, our value prediction technique successfully detects and removes 2.9% of all the faults, resulting in 3.3% of the faults to propagate outside the pipeline.

Figure 5.14 and Figure 5.15 show for issue queue and physical register file, the breakdown of all the faults regarding their effect on program execution. Both

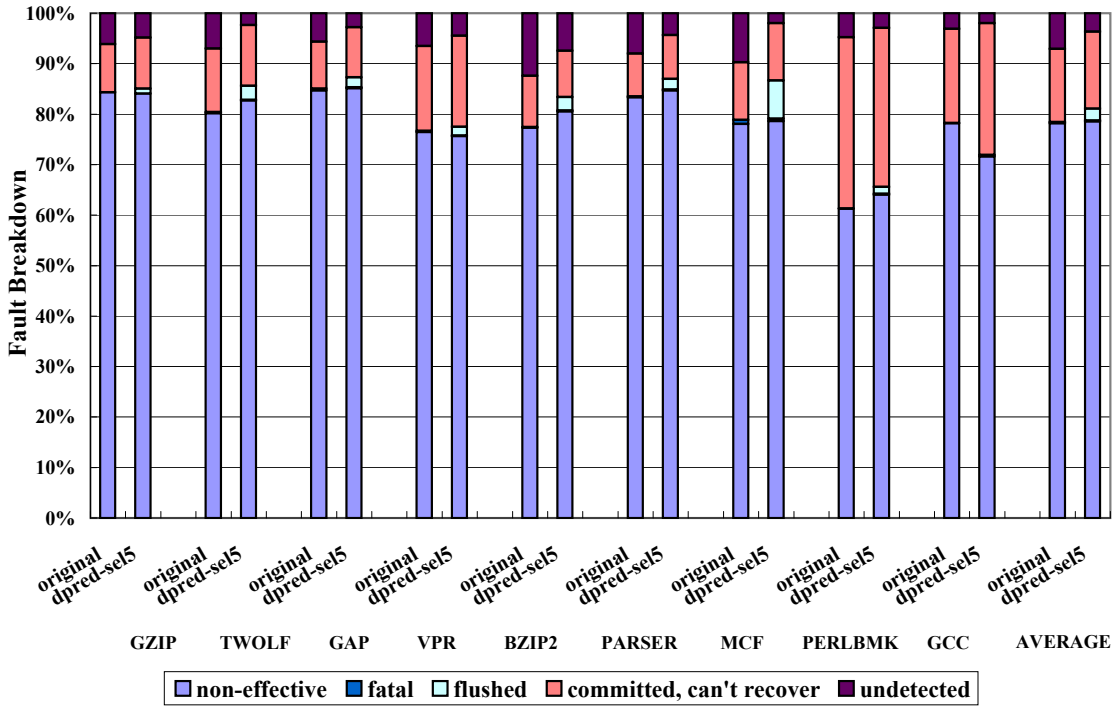

Figure 5.15: Breakdown of fault injections on physical register file by applying our selective value prediction technique. The minimum latency threshold for prediction is 5 cycles, measured from fetch to issue stage. Confidence threshold used for each prediction varies (high, medium or low threshold) according to the instruction's latency. On value misprediction, all the instructions in the pipeline are flushed and then re-fetch and re-execute. Categories include faults that have no architectural impact ("non-effective"), faults that cause program to crash or deadlock ('fatal"), faults that are removed during pipeline flushes before faulty instructions commit ("flushed"), faults that occur on registers while the most recent instructions updating the registers have committed ("committed, can't recover"), and faults that are not detected by value predictor before faulty instructions commit ("undetected").

the graphs are formatted similar to Figure 5.13, except that for physical register file, when faults occur on one register, it is possible the latest instruction which updates the corrupted register has committed, thus recovery–*i.e.*, flushing pipeline in our experiments–will not be able to restore processor state, thus we categorize such cases as "committed, can't recover". Comparing with the average results on fetch buffer which is shown in Figure 5.13, looking at program execution without value prediction, for issue queue and physical register file, respectively, on average

about 89.3% and 78.2% of all the faults occur on bits that have no impact on program execution, and about 2.8% and 0.2% of all the faults cause program crash or deadlock before the corrupted instructions commit, resulting in about 7.9% and 7.1% of the fault corruptions to propagate outside the pipeline (the "undetected" categorty)–about 14.5% of the fault corruptions on physical register file belong to "committed, can't recover" category. Once value prediction is enabled, about 2.9% and 2.4% of all the faults for issue queue and physical register file, respectively are now cleared off the pipeline by our technique (the "flushed" category), resulting in the portion of committed faults (labeled as "undetected") to be reduced to 3.7% for both issue queue and physical register file.

Figure 5.16 shows the overall MTTF which reflects the portion of faults that have impacted some instructions' computation while the corrupted instructions have still committed. In Figure 5.16, results on each hardware structure are reported separately. For each hardware structure, there are two groups of bars, representing original program run and program runs with value prediction. The MTTF results are reported as normalized to the original execution. From Figure 5.16, we see that relative to original execution, our value prediction technique improves the average MTTF across all the benchmarks to 6.23, 1.92 and 1.96 for fetch buffer, issue queue and physical register file, respectively.

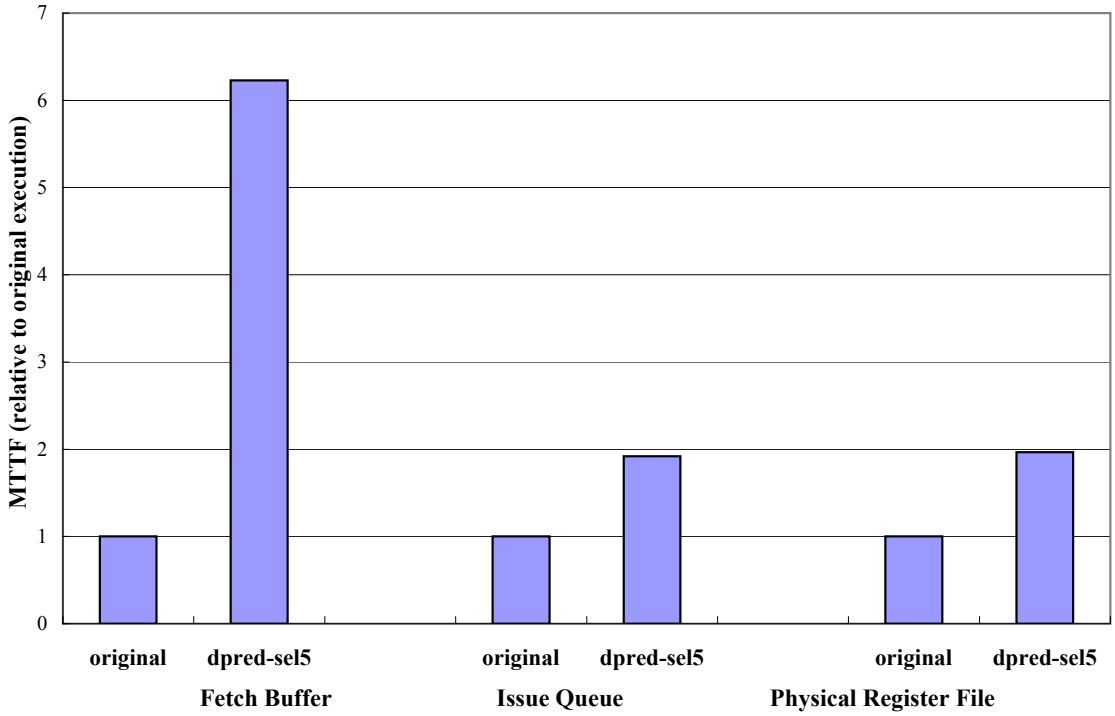

Figure 5.16: Average MTTF over 9 benchmarks in fetch buffer, issue queue and physical register file by applying our selective value prediction technique. The minimum latency threshold for prediction is 5 cycles, measured from fetch to issue stage. Confidence threshold used for each prediction varies (high, medium or low threshold) according to the instruction's latency. On value misprediction, all the instructions in the pipeline are flushed and then re-fetch and re-execute.

5.2.6 Discussion about Fault Injection and AVF Computation

As we have discussed, as two main methods in estimating system reliability. either fault injection or AVF computation has its own advantages and disadvantages. AVF computation is fast–it only requires one simulation run to obtain the reliability estimation of a processor design, but its results are conservative and can only provide a lower bound on the system reliability. Contrastingly, fault injection experiments can actually track fault propagation, and thus, provide an accurate estimation of the system reliability. However, it also requires a large number of samples, which usually means huge amount of computer time in order to get statistically accurate

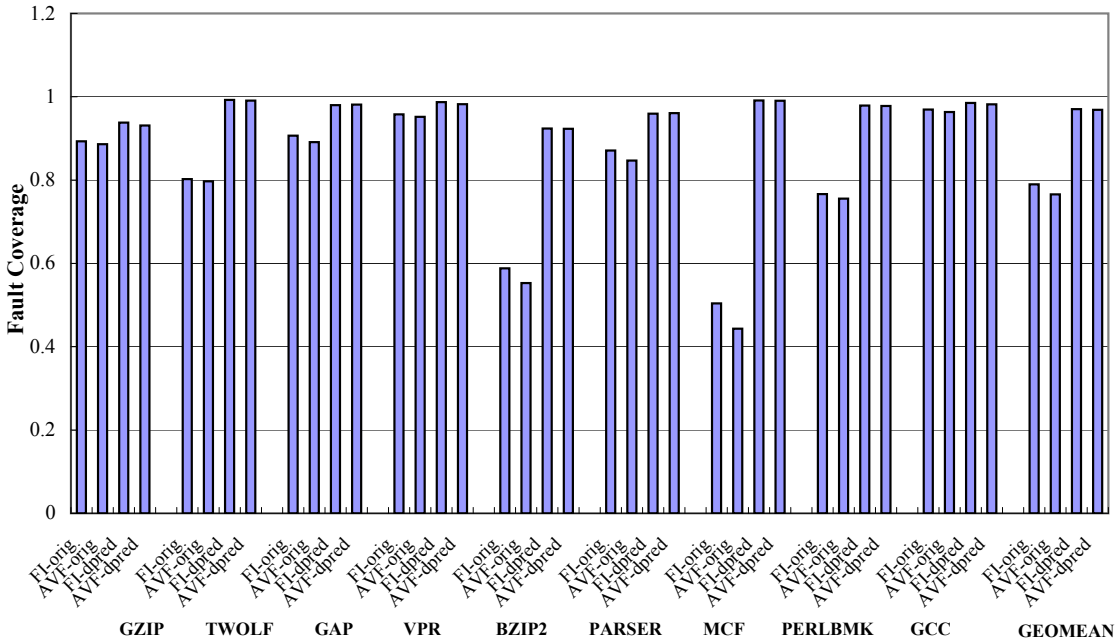

Figure 5.17: Fault coverage estimation for fetch buffer by fault injection experiments and AVF computation.

results.

Wang *et al* [38] analyzed and compared both methods. In our study, we have also employed both methods to estimate reliability impacts by ours or other fault detection technique. It is interesting to compare their experimental results.

Figure 5.17 reports for each benchmark, the fault coverage on fetch buffer estimated by fault injection experiments or AVF computation. To compute a hardware structure's fault coverage, given our fault injection results, we use the sum of the portions of "non-effective", "fatal" and "flushed" in Figure 5.13, Figure 5.14 and Figure 5.15; with AVF computation, we refer a hardware structure is invulnerable–covered from faults–at the probability of 1 minus AVF. In Figure 5.17, results on each benchmark are represented with one group of bars, while the last group is the average across all the benchmarks. In each group, the bars labeled "FI-orig"

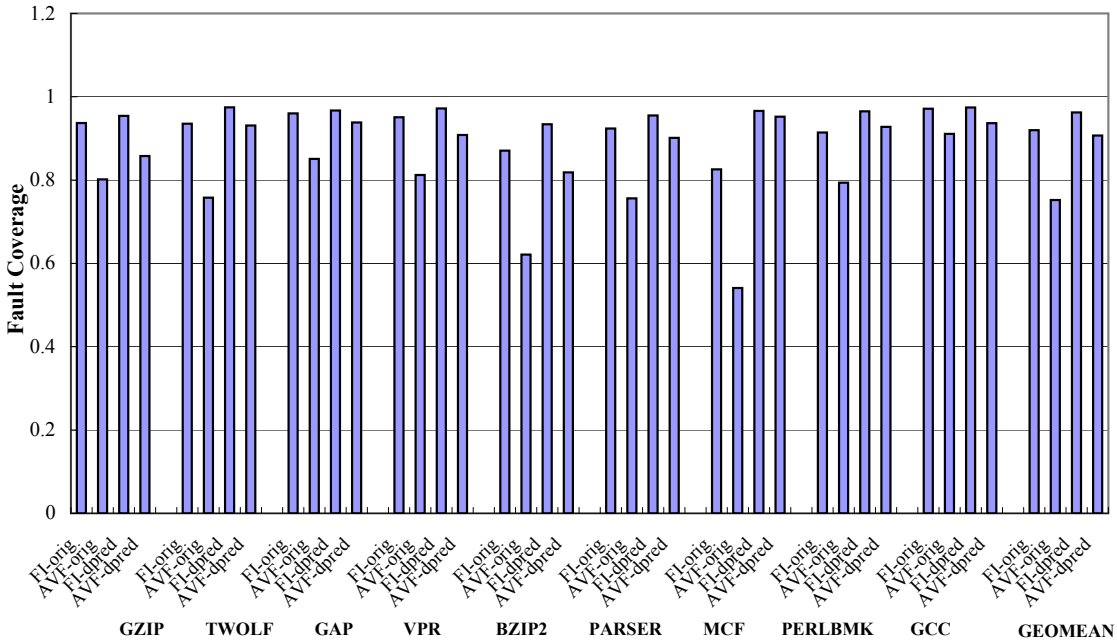

Figure 5.18: Fault coverage estimation for issue queue by fault injection experiments and AVF computation.

and "AVF-orig" exhibit the fault coverage results on original program execution by fault injection experiments and AVF computation, respectively, while the bars labeled "FI-dpred" and "AVF-dpred" exhibit the results on program execution with value prediction by fault injection or AVF computation. Figure 5.17 shows that generally, fault coverage estimated by the fault injection experiments is higher than AVF computation: on average, for original program execution and execution with value prediction, fault injection estimates about 79.0% and 97.0%, while AVF compuation estimates about 76.6% and 96.9%, respectively. Hence, AVF computation is 2.4% and 0.2% more conservative than using fault injections for program execution without and with value prediction, respectively.

Similarly, we report the fault coverage estimation on issue queue (Figure 5.18). Figure 5.18 shows that AVF computation results in even more conservative fault

coverage than fault injection experiments: on average, using fault injections results in 92.0% and 96.2%, while AVF computation results in 75.2% and 90.7% of faults to be covered for original program execution and execution with value prediction, respectively. Hence, for issue queue, AVF computation is 16.8% and 5.5% more conservative than fault injection campaign for program execution without and with value prediction, respectively. The bigger difference of fault coverage results on issue queue between fault injection and AVF computation is mainly due to the fact that AVF computation is hard to analyze within one simulation pass for every bit in a hardware entry, whether its corruption affects program execution or not; while the method of fault injection actually introduces fault into processor state and tracks fault propagation. Such disadvantage, exhibited as conservatism of estimated fault coverage, is more exposed on issue queue than fetch buffer. For issue queue, its entry contains more bits that are possible not to be used or affect instruction's computation–*e.g.*, it has one separate 26-bit immediate specifier, while for fetch buffer, the immediate data co-exists with other register tags.

5.3 Comparison with Fault Screening and Flush-on-L2-miss Techniques

This section compares our technique with fault screening, as well as the technique of flushing on L2 misses to reduce soft-error rate. Fault screening is proposed by Racunas *et al* [12], while flushing on L2 misses is proposed by Weaver *et al* [37]. To the best of our knowledge, fault screening is one of the most related work to

our study in exploiting program's inherent redundancy for fault tolerance. The technique of flushing on L2 misses is similar to ours in that both explore the selectivity on instruction vulnerability. In this section, Section 5.3.1 first summarizes their techniques and discusses the main difference from ours. Then, Section 5.3.2 compares performance cost of the three techniques, and Section 5.3.3 compares reliability gain using fault injection experiments. Lastly, Section 5.3.4 analyzes the sources of reliability gain.

5.3.1 Summary of Fault Screening and Flush-on-L2-miss Techniques

Racunas *et al* [12] proposed a technique called fault screening to detect value perturbation and prevent possible faults. Their technique tries to identify valid value space of an instruction's output, which is done by recording its past values as well as the value patterns. Future outputs that are not within the recorded space are considered as potentially corrupted. Among various implementations proposed in [12], the most practical one is called invariance-based screener. It works by recording how the bit values of instruction results change throughout program execution. Bits which keep the same values can be used to indicate future abnormal events such as the occurring of soft errors. More specifically, in [12], the authors keep track of bit invariance of memory instructions: a table with 1k bitmask entries is created for store/load addresses, and another 512-entry table for data stored to memory. Instruction address is used to index invariance table. Any change on the bitmasks triggers pipeline flush to eliminate and recover from potential soft errors.

The pipeline flush is achieved through branch recovery hardware. In addition, to reduce the effect of destructive aliasing, the bitmask tables are reset regularly during program execution.

It is very interesting to compare fault screener with our technique. The fault screener "predicts" possible value space. It is easy to implement, and more importantly, the screener is able to make prediction on most instructions–usually for most instructions, there always exist some bits that never change. However, its downside comes from the same fact that on average it only predicts/protects a portion of bit values, while unpredicted bits, which have changed values due to actual computation, are left for faults. On the contrary, our technique utilizes value prediction to try to find out the exact values–including all the output bits–within the whole data space. For an instruction output, all of its bits will be covered if the predictor can make prediction–as a result, compared to fault screener, value prediction can be more precise in capturing value discrepancy. Hence when detecting faults, if the predicted value space by fault screener is much smaller than the whole data space, it is very possible faults incur value perturbation and the screener catches them; but value prediction technique can perform better than the screener if corrupted values frequently fall outside the predicted space. Nevertheless, an advanced value predictor usually involves more hardware, and its prediction rate can not compete with bit-invariance screener either.

Weaver et al. [37] observed that the longer time an instruction spends in the pipeline, the more it is exposed to sources of soft errors such as neutron and alpha strikes, and hence, the more susceptible it becomes to faults. To reduce the time

instructions sit in vulnerable storage structures, they proposed to selectively squash instructions when long delays are encountered–more specifically, squash all the instructions following a load miss. Similar to their technique, our technique exploit the fact that instructions are different on their contribution to system vulnerability–we find a small portion of instructions account for a large fraction of system vulnerability. Furthermore, we quantify fault vulnerability at the instruction level, and only apply value prediction to those instructions that are most susceptible to faults–we evaluate an instruction's vulnerability by its latency from fetch to issue stage–and trigger recovery on mispredictions. Therefore, our technique covers more long-latency instructions than load misses, and hence, better improves system reliability with small additional performance degradation.

In addition, all the three techniques do not guarantee failure-free–it is hard for them to detect or remove all possible faults. Even worse, both our technique and fault screener may claim faults that do not exist ("false positive"): *i.e.*, for value prediction, every misprediction presents a false positive; for fault screening, every natural bit-variation presents a false positive. For Weaver's technique, it triggers pipeline flush on L2 cachemiss no matter whether faults have actually occurred or not. Thus, program performance is degraded when recovery is not necessary, which is another important issue besides reliability impact. But as we have discussed previously, because we selectively predict instructions that have long residence time in pipeline, re-execution during recovery on misprediction usually completes faster than the original run, which leads to shorter exposure time of re-executed instructions to faults and thus improves program reliability indirectly. Our experimental

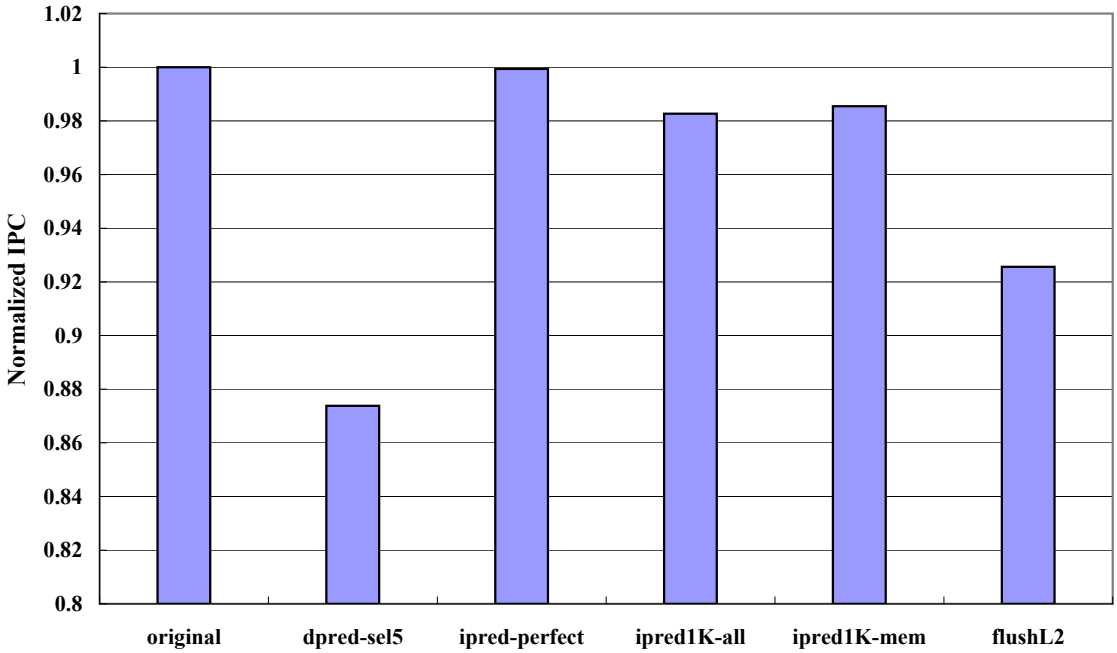

Figure 5.19: IPC (relative to original execution) on 9 spec2000 integer benchmarks with either value prediction, bit-invariance screening, or flushing on L2 miss implemented. On value or bit-invariance misprediction, the whole pipeline is flushed and then re-fetch after a 3-cycle penalty. On L2 miss, all the instructions following the load miss are squashed from the pipeline and they start to be re-fetch after the cache miss is resolved.

results below also show the reliability benefits from recovery for both techniques.

5.3.2 Performance Impact

Before we evaluate how the three techniques detect and recover from faults, we examine their impacts on program performance. We use the same detailed architectural simulator described in Section 5.2. The simulator settings are also the same as listed in Table 3.1. We set our value predictor so that it selectively predicts instructions that stay equal to or more than 5 cycles in pipeline from fetch to issue stage. As for confidence estimation, we adopt the set of latency threshold (5, 10, 20) to associate with the highest, medium, and lowest confidence threshold, sepa-

rately (related analysis is discussed in Section 5.2.3). In addition, we implement the bit-invariance screener as [12], using two tables to store value bitmasks, one for store/load addresses, another for store data. Each bitmask table has 1K entries and each entry is 32 bits long. The screener only makes predictions on store/load instructions. For better investigation, we also implement two other forms of screeners: one has infinite table-entries and predicts not only store/load instructions, but also other instructions that produce outputs; the other one has limited table size–1K entries, and it predicts both memory instructions and all other result-computing instructions. In our experiments, to be consistent with the configuration in [12], on each value misprediction or change in invariance bits, the whole pipiline is flushed and program starts to re-fetch and re-execute from the top of the flushed instructions. In the experiments, we assume a 3-cycle penalty from when a misprediction is detected until the first re-fetched instruction can enter the pipeline. To implement the flush-on-L2-miss technique, we mark a load instruction if it incurs L2 miss once executed; the flag of each load instruction is checked in each processor cycle–checking starts from the oldest load in the pipeline–and if a flag set is detected, all the subsequent instructions after the load instruction are flushed from the pipeline; the flushed instructions will start to be re-fetched after the cachemiss is resolved.

Figure 5.19 reports for the benchmarks listed in Table 5.2, the average IPC of their basic runs and those with value prediction, bit-invariance screening, or flushing on L2 miss. In Figure 5.19, each group of bars represents experiments on one benchmark, while the last group represents the geometric mean across all the benchmarks. In each group, the bar labeled "original" represents IPC of the original execution;

the bar labeled "dpred-sel5" indicates the IPC of execution with value prediction; the other 3 bars are for experiments with bit-invariance screeners: the bar labeled "ipred-perfect" reports IPC of execution with infinite-table-size screener; the bar labeled "ipred1K-all" reports IPC of execution with 1K-entry-table screener–again, experiments on both "ipred-perfect" and "ipred1K-all" predict memory instructions and all other result-computing instructions; the next bar labeled "ipred1K-mem" is for execution with 1K-entry-table screener and only predicts memory instructions, as implemented in [12]; at last, the bar labeled "flushL2" exhibits IPC of execution which flushes the pipeline on L2 miss. All the results in each group of bars are normalized by the corresponding IPC of original run (hence the bars labeled "original" are always 1).

The results show that program execution with value prediction runs the slowest– on average, our technique degrades program performance by about 12.6%, while fault screening technique results in much less performance cost: screening with infinite table has about 0.1%, screening with 1K-entry table and predicting all memory and other output-computing instructions has about 1.7%, screening with 1K-table and only predicting memory instructions has about 1.5% degradation. For execution in which L2 misses trigger pipeline flushes, the performance is degraded by 7.4%. The bigger performance degradation of our technique is mainly due to the larger number of pipeline flushes caused by value mispredictions.

5.3.3 Fault Injection Results

As for reliability evaluation, unfortunately, it is hard to compute AVF for execution with bit-invariance screening. This is because the screener only covers a portion of output bits, which cannot indicate the correctness status of other bits in processor such as control bits. Hence in our study, we implement fault injection experiments to capture the techniques' capability on fault detection.

In addition, [12] used a functional simulator to inject faults–faults are randomly injected to all instructions' output bits, then those that are caught by the screener are recorded. We claim functional simulator is too simple for accurate fault injection experiment. For example, as we have discussed before, the longer an instruction stays in processor, the more chances it is exposed to faults. Functional simulation does not incorporate such timing effect. Hence we conduct fault injection experiments on the detailed architectural simulator as used in Section 5.3.2. Furthermore, we follow the same fault injection methodology as in Section 5.2.5, and study reliability impact on the three hardware structures: the physical register file, the fetch queue, and IQ.

Similar to the fault injection experiments in Section 5.2.5, for all the injected faults, if they corrupt valid processor state, we track their propagation until faulty instructions have been flushed or commit, or program crashes. Figure 5.20 breakdowns all the faults on the three hardware structures–fetch buffer, IQ and physical register file–by their effects on program execution. For each benchmark, we report fault injection results from 6 program runs on each of the three hardware structures–

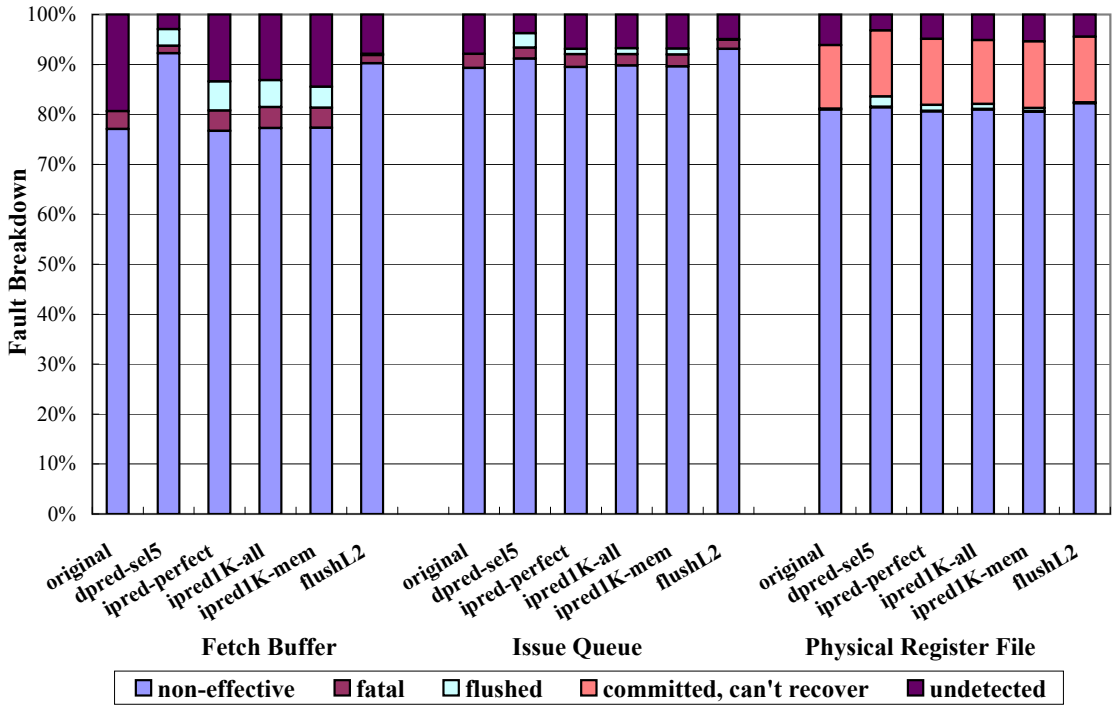

Figure 5.20: Breakdown of fault outcomes on fetch buffer, issue queue and physical register file by applying value prediction, fault screening or flush-on-L2-miss techniques. Categories include faults that have no architectural impact ("non-effective"), faults that cause program to crash or deadlock ('fatal"), faults that are removed during pipeline flushes before faulty instructions commit ("flushed"), faults that occur on physical register file but the latest instruction which updates the corrupted register has committed ('committed, can't recover"), and faults that are not detected by value predictor or bit-invariance screener before faulty instructions commit ("undetected").

represented with a group of 6 bars, respectively: original program execution, program execution with value prediction, program execution with fault screening that has infinite table and predicts memory instructions as well as other instructions producing outputs, program runs with fault screening that has 1K-entry table and also predicts both memory and other result-computing instructions, program runs with fault screening that has 1K-entry table and only predicts memory instructions, and program execution with flush-on-L2-miss technique. Similar to Figure 5.15, each bar contains five categories, labeled "non-effective", "fatal", "flushed", "com-

mitted, can't recover" and "undetected", respectively. Each category represents the same class of fault injections as in Figure 5.15. From Figure 5.20, we see that fault detection techniques successfully remove some number of faults from the pipeline: for the three structures–fetch buffer, IQ and physical register file, value prediction technique detects and removes 3.3%, 2.9% and 2.4%, respectively of all the faults; fault screening with infinite table and predicting both memory and all other result-producing instructions removes about 5.8%, 1.0% and 1.4%, respectively; fault screening with 1K-entry table and predicting all memory and output-producing instructions removes about 5.4%, 1.2% and 1.2%, respectively; fault screening with 1K-entry table and only predicting memory instructions removes 4.2%, 1.2% and 0.7%; while flush-on-L2-miss technique removes about 0.2%, 0.1% and 0.1%, respectively. As a result, in execution with fault detection techniques, less percentage of faults can propagate out of the pipeline: for fetch buffer, IQ and physical register file, the "undetected" portion is 19.3%, 7.9% and 7.0%, respectively for original run; 2.9%, 3.7% and 3.7%, respectively for execution with value prediction; 13.4%, 6.9% and 5.6%, respectively for execution with fault screening which has infinite table and predicts both memory and all other result-producing instructions; 13.2%, 6.8% and 5.9%, respectively for execution with fault screening which has 1K-entry table and predicts all memory and output-producing instructions; 14.5%, 6.8% and 6.2%, respectively for execution with fault screening which has 1K-entry table and only predicts memory instructions; 7.9%, 5.0% and 5.1%, respectively for execution with flush-on-L2-miss technique. In addition, for all the faults on physical register file, considering the part of the fault injections that occur on registers for which

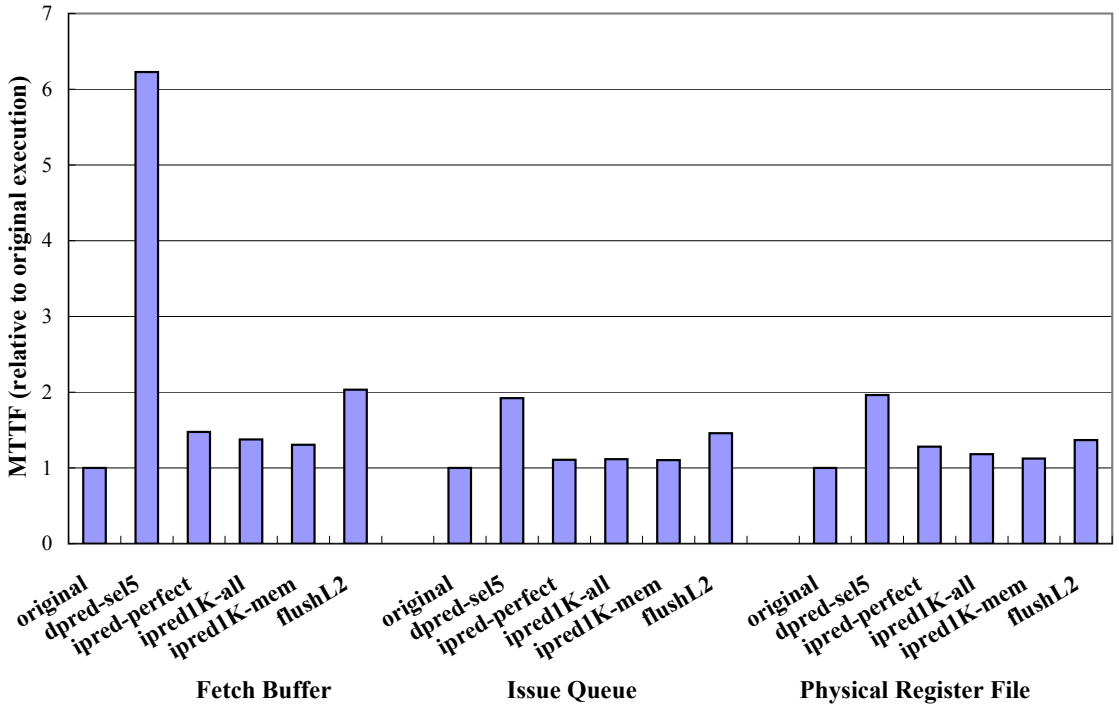

Figure 5.21: Average MTTF over 9 benchmarks in fetch buffer, issue queue and physical register file by applying value prediction, fault screening, or flush-on-L2-miss techniques.

the latest instructions that have updated the corrupted registers have committed, the portion of faults that cannot be recovered by the fault detection techniques is now 21.6%, 18.9%, 20.9%, 20.5%, 21.6% and 20.3% for original execution, execution with value prediction, fault screening with infinite or 1K-entry table and predicting both memory and all other result-producing instructions, fault screening with 1K-entry table and only predicting memory instructions, and execution with flush-on-L2-miss technique, respectively. In all, our value prediction technique removes the largest percentage of the injected faults, and hence results in the fewest faults that propagate outside the pipeline.

With the faults' breakdown shown in Figure 5.20, we report the current MTTF–the percentage of faults that impact program execution but cannot be suc-

cessfully detected and removed from pipeline by fault detection techniques–in Figure 5.21. In Figure 5.21, results on each hardware structure are reported separately. For each hardware structure, there are six groups of bars, representing six different program execution including original program run, and program execution with either value prediction, fault screening or flush-on-L2-miss technique as we have mentioned. For each type of program execution, there are two bars representing average MTTF results across all the benchmarks of MTTF–MTTF results are normalized to the original execution. Figure 5.21 shows that value prediction performs the best in improving program reliability: for fetch buffer, IQ and physical register file, value prediction improves their MTTF by about 6.23, 1.92 and 1.96, respectively compared to the original execution; fault screening with infinite and predicting both memory and all other result-producing instructions improves by about 1.48, 1.11 and 1.28, respectively; fault screening with 1K-entry table and predicting both memory and all other result-producing instructions improves by about 1.37, 1.11 and 1.18, respectively; fault screening with 1K-entry table and only predicting memory instructions improves by about 1.31, 1.10 and 1.13, respectively; flush-on-L2-miss technique improves by 2.03, 1.46 and 1.37, respectively.

5.3.4 Fault Detection Analysis

Although bit-invariance screener may make prediction on more instructions than value predictor, the latter performs better in our fault injection experiments. This is due to various reasons. For example, instructions computing with immedi-

ate values usually have more variance bits in their outputs–in other words, those instructions have bigger value space to be predicted by bit-invariance screener, which makes it harder for the screener to detect faults. Moreover, for fetch buffer and issue queue, their immediate fields constitute big portions of each entry, thus appear more vulnerable to faults. Compared to bit-invariance screener, value predictor performs more sensitively: any difference between prediction and computation results is considered as symptom of potential fault.

Another big source of fault detection by value prediction comes from the fact that value predictor has lower prediction rate than bit-invariance predictor–faults can be removed from pipeline by flushes caused by natural mispredictions. Although such mispredictions also degrade program performance, the performance loss can be justified by the reliability benefit because our technique selectively targets on long-latency instructions.

Compared to flush-on-L2-miss technique, value prediction performs better because it covers more long-latency instructions than load misses. Prediction or flushing on those vulnerable instructions improves system reliability while incurring small additional performance degradation.

Chapter 6

Related Work

This work is related to the following categories of research.

6.1 Soft Computation

Several researchers have studied soft computations. Breuer [39, 40] recognized multimedia workloads can tolerate errors, and proposed exploiting this error resilience to address manufacturing defects. Our definition of application-level correctness is similar to Breuer's notion of "error tolerance" (ET) [39], which allows chips that produce numerically incorrect results to be usable as long as their results are acceptable to the users. The main difference is that Breuer exploited ET to tolerate hardware defects for higher chip yield, whereas we identify application-level correctness as one level of redundancy inherent to applications, and exploit it to tolerate transient faults assuming all the hardware is functionally correct. Another difference is that we further quantify the notion of "degree of acceptability" with appropriate fidelity metrics and fault injection experiments to directly measure user satisfaction.

Other soft computing research includes [41] by Liu *et al* in which they observed certain image processing and tracking algorithms can be inexact. They exploited this fact to improve task schedulability in real-time systems. Palem [42, 43] studied

probabilistic algorithms to build randomized circuits that are extremely energy efficient. Finally, Alvarez and Valero [44, 45] found that multimedia applications are resilient to precision loss. They developed novel value reuse techniques for floating point operations so that energy consumption is reduced greatly. In addition, Wang et al. [19] identified some outcome-tolerant branches–called "Y branches". They presented the performance speedup by removing mispredictions on those branches. Compared to all the previous studies, we exploit soft computations for enhancing program reliability.

6.2 Fault Susceptibility Characterization

In characterizing soft error susceptibility, there are a great number of prior studies to which our work is related. Among them, the most related are researches which used detailed CPU models and measured the effects of soft errors by injecting faults into their models. For example, Saggese *et al* [46] injected faults into a DLX-like embedded processor; Wang et al. [47] injected faults into a CPU similar to the Alpha 21264 or AMD Athlon; Kim and Somani [48] injected faults into Sun's picoJava-II; and Czeck and Siewiorek [49] injected faults into an IBM RT PC processor. All of these fault susceptibility studies used gate- or RTL-level models, and injected faults into the entire CPU. In contrast, our study uses a detailed architecture model, and focuses fault injections on important hardware structures such as physical register file, fetch queue, and issue queue.

Another main difference between our work and all previous studies on soft

error susceptibility is the definition of correctness used to evaluate the effects of soft errors. Previous work requires architectural state to be numerically exact for program execution to be correct; in contrast, our work only requires program outputs to be acceptable to the user. By evaluating correctness at a higher level of abstraction, we measure the *additional* portion of soft errors that can lead to acceptable program outputs. In addition, such notion of application-level correctness depends on the characteristics of different applications, and thus needs specific fidelity metrics for evaluation. It is also user-dependent for determining the amount of acceptable errors in answer quality. All these distinguish our study from previous approaches.

6.3 Analysis of Fault-Tolerance Sources

Another related area to our work is in analyzing sources of fault-tolerance. Previous research has noticed that not all faults result in visible effects. Shivakumar *et al* [50] studied masking at the *circuit level*. They developed an electrical and latching-window masking model, and predicted the impact of these circuit effects on soft error rates. Kim *et al* [51] studied logical masking. They propose "Susceptibility Tables" for logic gates that model the probability a soft error will propagate through a combinational logic block. Mukherjee *et al* [18] identified *microarchitecture-level masking* (mispeculated instructions, predictor structure bits, and microarchitecturally idle bits) as well as *architecture-level masking* (NOP instructions, performance-enhancing instructions, dynamically dead code, and logically masked instructions). Wang *et al* [19] observed that certain conditional branch

outcomes can be wrong without affecting program correctness ("Y branches"), which is another form of architecture-level masking.

Our work differs mainly in that we take algorithm-level resilience into consideration, such as the existence of multiple valid outputs and user-level interpretation. We also explore algorithmic properties, such as redundancy, adaptivity, and reduced precision, which greatly enhance program's capability to tolerate errors. Such resilience exploration is helpful for both architecture- and compiler- design.

6.4 Fault-Tolerance Techniques

Traditional techniques like Error Correcting Code (ECC) and parity bits have been widely adopted to protect various hardware structures, especially memory units. However, these techniques are too costly and thus impractical to implement on all logic units. Currently, researchers focus on developing new fault-tolerance techniques to achieve effective fault detection/recovery, or improve system fault susceptibility.

6.4.1 Fault Detection

To detect or recover from faults, designers typically introduce or utilize explicit redundant execution in hardware. Faults are detected by comparing results from two copies of program execution. For example, Horst *et al* [3] used one separate processor, while Austin [52] used an additional in-order processor, to recompute and verify the computation results of the main out-of-order processor. To exploit the exist-

ing architecture redundancy, Rotenberg [53] suggested simultaneous multithreading (SMT) platform, while Sundaramoorthy and Purser [54] used chip multi-processor (CMP) to execute the additional copy of program. Ray *et al* [6] proposed another mechanism which relies on register renaming hardware to temporarily split instruction stream into multiple threads, and then verify their results.

To avoid expensive hardware cost, compiler-based approaches duplicate instructions on the software side, such as EDDI proposed by Oh *et al* [9] and SWIFT by Reis *et al* [10]. To reduce the performance cost, Reinhardt and Mukherjee [5] proposed an improved SMT-based approach which only checks instructions whose side-effects exit the processor core. Reis *et al* [23] proposed to combine both hardware and compiler based approaches for better tradeoff between reliability and hardware cost.

Despite different implementation details, for both hardware and software approaches, extra redundancy is explicitly created–or existing redundancy is exerted in addition to original program execution, which usually involves expensive performance or hardware cost. In our work, we exploit program's inherent redundancy such as value predictability for redundant execution. By exploiting such inherent redundancy, our technique achieves remarkable reliability improvement with much small performance degradation.

Regarding to exploiting program inherent redundancy, one of the most related work to ours is [12]. As described in Section 5.3.1, Racunas *et al* made use of value perturbation to detect possible faults. Their technique tries to identify valid value space of an instruction, which is done by keeping track of past results of that

instruction. Future output that is not within the recorded value (pattern) space is considered as potentially corrupted. Compared to their technique, we exploit value predictability to enhance program reliability. The main difference between value perturbation and value prediction is that value prediction tries to predict an instruction's result exactly. Output that is not equal to the predicted value is considered as potentially corrupted. Although detecting value perturbation seems easier, value prediction can be more precise in finding discrepancy. For example, an instruction's past value space can be so big that corrupted values by faults may still fall in valid value space, and thus can't be detected by value perturbation technique. Our experiments in Section 5.3.3 compare the reliability impacts of both techniques, and the results show the effectiveness of our technique.

6.4.2 Fault Recovery

Fault recovery is usually implemented as a complement to fault detection since to recover from faults, they have to be detected first. Thus, although the performance impact of fault recovery techniques is not crucial considering the fact that they are incurred very infrequently, researchers still work hard to develop efficient mechanisms. For example, Active-stream/Redundant-stream Simultaneous Multithreading (AR-SMT) proposed by Rotenberg [53], can achieve recovery since the committed state of the redundant stream can be used as a checkpoint. Vijaykumar *et al* [32] extended their SRT(Simultaneously and Redundantly Threaded) technique for fault detection with recovery scheme. Their modifications included buffering in-

structions from committing until the instructions' outputs have been verified, plus methods to avoid stalling instructions at commit while waiting for verification. Similarly, Gomma *et al* [55] proposed hardware-assisted fault recovery for CMPs on the basis of their Chip-level Redundantly Threaded (CRT) technique for transient-fault detection. Compared to their techniques, our checkpoint recovery mechanism is advantageous as we only save part of program state thus performance cost incurred is very small.

Our recovery mechanism also relates to the vast research area on checkpointing. Checkpointing is widely used in systems such as parallel & distributed computing. For example, Chandy and Lamport [56] proposed a global snapshot algorithm for distributed systems, which is widely adopted and extended to minimize the overheads of coordination and context saving by a large number of studies, such as the work by Kim and Park [57]. Ahmed *et al* [58] proposed to checkpoint process context and global state based on visible cache line modifications for shared-memory systems. Compared to their approaches, we focus on only checkpointing necessary state for application-level correctness in uniprocessor systems. Thus our checkpoint is very cheap but still effective–our lightweight recovery technique can allow a large number of faults to be successfully recovered while our soft-checkpointing technique can recover most of them.

6.4.3 Reducing Fault Susceptibility

Besides designing more efficient techniques to detect and recover from faults, researchers have also realized that reducing, but not eliminating, soft-error rate to achieve less performance degradation is more desirable in some cases. For example, many systems, such as commodity computers, do not need full or perfect coverage. Based on this observation, Weaver *et al* [37] proposed techniques to reduce error rate by either squashing instructions when long delays are encountered, or delaying to signal faults until the corresponding instructions are determined not to be dynamically dead code. Gomma and Vijaykumar [59] adopted similar approach but on an SMT platform, which only triggers redundant thread for fault detection during low-ILP and L2 cache misses. They also proposed to detect faults during high-ILP by instruction reuse.

Although their techniques are effective in reducing system vulnerability to faults to some extent, the way they determine when to enable fault protection is only by monitoring program performance. On the contrary, our checkpointing mechanism discounts computations that are inherently resilient to errors, while our fault detection technique exploits value predictability for redundant execution and takes value misprediction as symptom of potential faults, thus have much less overhead on both hardware and performance.

In addition, our technique considers fault vulnerability at the instruction level, which is absent from most existing techniques. By quantifying instruction's vulnerability, we selectively protect instructions that are most susceptible to faults, thus

reduce mispredictions and the related recovery cost, while still maintaining acceptable reliability budget.

6.5 Symptom of Potential Faults

Wang *et al* [11] proposed the concept of fault symptom–hint at the presence of soft errors. They exploited symptoms–such as branch misprediction and cache misses–to trigger fault recovery mechanism. Similarly, Racunas *et al* made use of value perturbation to prevent possible faults. Perturbation from the recorded value space of an instruction is viewed as caused by soft errors, thus triggers corresponding recovery mechanism. Similar to their work, we consider value misprediction as implication of potential faults, and take recovery action–such as flushing pipeline in our experiments–to try to remove and recover from faults. The main difference between their work and ours is that we view value predictability as inherent redundancy for comparing computational results. We also characterize fault vulnerability at the instruction level, and propose selective prediction to reduce mispredictions and the subsequent performance cost.

Chapter 7

Conclusions

In this chapter, we first summarize the whole dissertation (Section 7.1), then we enumerate our contributions to fault tolerance research (Section 7.2), and lastly, we propose possible directions for future study (Section 7.3).

7.1 Summary and Conclusion

This work exploits program's inherent redundancy for enhancing fault tolerance. First, we investigate additional fault resilience at the application level. We explore definitions of program correctness that view correctness from the application's standpoint rather than the architecture's standpoint. Traditionally, correct program's execution requires architectural state to be numerically perfect. However, in many cases, even if program execution is not 100% numerically correct, it may be completely acceptable if the answers can satisfy the user's requirement. Hence, faults which have caused such numerically faulty execution are no longer intolerable– programs appear to be more tolerant at the user(*i.e.*, application) level. We conduct fault injection experiments and measure the additional fault tolerance at the application level compared to the traditional architecture level. Our results show for soft computations, about 45.8% of fault injections that lead to architecturally incorrect execution are correct under application-level correctness. We also exploit the re-

laxed requirements of application-level correctness to reduce checkpoint cost: our lightweight recovery mechanism checkpoints a minimal set of program state, but can successfully recovers a major part of program crashes in soft computations; our soft-checkpointing technique identify computations that are resilient to errors and excludes their outputs from checkpoint, thus can successfully recovers almost all of program crashes in soft computations.

We also investigate another form of redundancy inherent in program–value predictability. We take value prediction as additional program execution and compare with actual computation results–misprediction is considered as symptom of potential faults. To reduce misprediction rate caused by limitations of predictor itself, we characterize fault vulnerability at the instruction level and only apply value prediction to instructions that are highly susceptible to faults. We also vary threshold of confidence estimator according to instruction's vulnerability–instructions with high vulnerability are assigned with low confidence threshold, while instructions with low vulnerability are assigned with high confidence threshold. Our results show large reliability gain with very small performance degradation of our selective prediction mechanism.

7.2 Contributions

This dissertation makes the following contributions within the context of exploiting program's inherent redundancy to enhance fault tolerance.

I. Traditional fault tolerance studies adopt strict correctness definitions and re-

quire perfect numerical integrity of program execution. Such strict requirements ignore the flexibility to numerical values from user's point of view, thus cause overdesign of systems. Our work explores the numerical redundancy at the application level–*i.e.*, the existence of multiple numerical outputs which can all be accepted by users. Such redundancy provides additional fault tolerance: faults that cause program to produce numerically different but acceptable outputs are also tolerable to users. We implement fault injection experiments on a detailed architectural processor model, and measure the additional fault tolerance at the application level, compared to the traditional architectural level.

II. In our study of application-level fault tolerance, we mainly examine soft computations including multimedia and AI. These areas have been rapidly developing and widely applied to our modern society. Our analysis shows there are various sources of redundancy originated in the applications' own characteristics. Such algorithmic exploration helps understand the behavior of programs in face of faults, and exposes more chances for cost-effective system design.

III. We implement new fault recovery techniques by exploiting the additional redundancy at the application level. One technique we propose is lightweight fault recovery which only checkpoints a minimal set of program state–program counter, architectural register file, and stack. Another technique first identifies computations that are resilient to errors, and then excludes state that store their outputs from checkpointing. Such checkpointing mechanisms try to only

save necessary state for fault recovery, and mainly rely on programs themselves to absorb fault corruption and still generate acceptable outputs. Thus the cost is much smaller than the traditional full checkpointing mechanisms, while our results show they are effective in fault recovery.

IV. In additional to studying application-level correctness, we study value predictability as another source of redundancy for fault tolerance enhancement. We apply value prediction to check computational results and detect potential faults–we take value misprediction as symptom of fault occurring, and use re-execution to try to remove faults from pipeline. By exploiting the inherent redundancy in program–value predictability, we avoid expensive hardware or performance cost by introducing redundant execution explicitly, which is widely adopted by current fault tolerance research.

V. In order to reduce performance cost caused by additional fault recovery, we characterize instruction's vulnerability by computing the percentage of a hardware structure's average AVF that an instruction relates to. We find that a small portion of instructions accounts for a major fraction of program vulnerability. Thus, by selectively protecting such a small portion of instructions from fault corruption, the overall reliability can be greatly improved greatly with relatively much less performance cost than full protection of all instructions. We also exploit such variation in instruction vulnerability to confidence estimation of value prediction, and propose adaptive confidence threshold to better trade off reliability gain and performance cost.

7.3 Future Directions

In this dissertation, we show it is effective to reduce fault rate by exploiting program's inherent redundancy. We implement fault injection experiment and measure the additional fault tolerance provided by the relaxed correctness definition at the application level. We also investigate the potential of utilizing value predictability to check computational results and detect possible faults. We believe the ideas presented in this work can be further studied.

I. In this dissertation, we study application-level correctness as well as the additional fault tolerance it provides. We also implement cost-effective checkpointing mechanisms in exploiting such numerical redundancy–*i.e.*, multiple numerical outputs can appear to be acceptable from user's standpoint. There exist more opportunities in this direction. For example, for fault detection, it is not necessary to monitor the parts of program execution that are resilient to errors since fault corruption on those parts can be absorbed by program itself and acceptable outputs, although numerically different from fault-free execution, can still be produced. This can be achieved through either hardware (e.g., utilizing additional hardware thread or processor) or software (e.g., inserting additional check code during compiler stage) approaches. Such mechanisms can be more efficient and have less impact on program performance, compared to protecting the whole program.

   However, more practical ways of identifying fault-resilient computations or program state have to be studied more carefully. In our soft-checkpoint exper-

iment, we establish the set of soft state manually–first mark computations that are resilient to errors, then track the state that store the corresponding output values. This has to be transformed to more automatic methods. For example, with some assistance from programmers, compilers may convert program code and incorporate such knowledge about state softness.

In addition, the information on application-level fault tolerance can also be applied to other fields such as compiler optimization. One key observation is that the longer one instruction stays in the pipeline, the more vulnerable it becomes [37]. On the other hand, our analysis shows that instructions which process soft data, *e.g.*, approximate data, are tolerant to faults. Therefore, during program compilation, the compiler can transform code in a way that instructions unrelated to soft data are optimized to stay the least amount of time in the processor, while the remaining instructions can be scheduled less efficiently. Moreover, because soft and non-soft data are often interleaved in the memory, program can be scheduled so that instructions requesting memory access for soft data are executed first and incur cache misses shortly, while the following instructions for non-soft data will suffer little cache-miss penalty since the shared memory blocks have been brought into cache. Thus, the portion of execution time spent on fault-resilient computations is increased, and program's overall reliability is enhanced.

II. In our study, we characterize instruction's vulnerability and find the majority of program's overall reliability relies on a small portion of executed in-

structions. We exploit such observation in our experiments to reduce value misprediction rate as well as the accompanying recovery cost. Such idea of selective protection can be extended to other studies in which reliability and performance cost can be traded by shifting more efforts to instructions that are more susceptible to faults. For example, a redundant thread that is created for checking computational results only needs to re-execute when program proceeds slowly since that is the time instructions in pipeline are exposed to faults for longer time and thus become more susceptible.

III. As we have discussed, flushing pipeline can possibly remove faults from propagating to memory and becoming non-recoverable. Accompanied with the reliability benefit is performance degradation by re-fetching and re-executing instructions. However, if problems that cause program to run slowly–such as cache misses or shortage of computation resources–have been resolved during re-execution, program can then proceed faster than its original run, and appear to be less vulnerable than without flushing. Thus it is possible performance loss by flushing is made up with reliability gain, which can be perceived by metrics that incorporating both reliability and performance such as MITF. Therefore, it is interesting to explore more deeply the relations between flushing and MITF benefit–*i.e.*, identifying the set of instructions which can bring the most MITF benefit if flushed.

Such study is also interleaved with our work on exploiting value predictability. Predicting an instruction can benefit reliability without any performance loss,

but has to pay other price such as power or more hardware for better prediction accuracy. On the other hand, flushing an instruction may or may not benefit the compromise between reliability and performance (i.e., MITF). Thus it will be very useful to incorporate instruction's predictability, together with the impact on MITF by flushing, into the policy of selective protection.

Bibliography

[1] B. Calder, G. Reinman, and D. Tullsen, "Selective Value Prediction," in *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pp. 64–74, May 1999.

[2] S. S. Mukherjee, J. Emer, and S. K. Reinhardt, "The soft error problem: an architectural perspective," in *Proceedings of the 2005 IEEE 11th International Symposium on High Performance Computer Architecture*, pp. 243–247, February 2005.

[3] R. W. Horst, R. L. Harris, and R. L. Jardine, "Multiple instruction issue in the NonStop Cyclone processor," in *Proceedings of the 17th International Symposium on Computer Architecture*, pp. 216–226, May 1990.

[4] Y. Yeh, "Triple-triple redundant 777 primary flight computer," in *Proceedings of the 1996 IEEE Aerospace Applications Conference*, pp. 293–307, February 1996.

[5] S. K. Reinhardt and S. S. Mukherjee, "Transient Fault Detection via Simultaneous Multithreading," in *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pp. 25–36, June 2000.

[6] J. Ray, J. C. Hoe, and B. Falsafi, "Dual use of superscalar datapath for transient-fault detection and recovery," in *Proceedings of the 34th annual*

IEEE/ACM International Symposium on Microarchitecture, pp. 214–224, December 2001.

[7] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt, "Detailed design and evaluation of redundant multithreading alternatives," in *Proceedings of the 29th annual International Symposium on Computer Architecture*, pp. 99–110, May 2002.

[8] N. Oh, P. P. Shirvani, and E. J. McCluskey, "Control-flow checking by software signatures," in *IEEE Transactions on Reliability*, pp. 111–122, March 2002.

[9] N. Oh, P. P. Shirvani, and E. J. McCluskey, "Error detection by duplicated instructions in super-scalar processors," in *IEEE Transactions on Reliability*, pp. 63–75, March 2002.

[10] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, "SWIFT: Software implemented fault tolerance," in *Proceedings of the 3rd International Symposium on Code Generation and Optimization*, pp. 243–254, March 2005.

[11] N. Wang and S. J. Patel, "ReStore: Symptom Based Soft Error Detection in Microprocessors," in *Proceedings of the 2005 International Conference on Dependable Systems and Networks*, pp. 30–39, June 2005.

[12] P. Racunas, K. Constantinides, S. Manne, and S. S. Mukherjee, "Perturbation-Based Fault Screening," in *Proceedings of the 2007 IEEE 13th International Symposium on High-Performance Computer Architecture*, pp. 169–180, February 2007.

[13] X. Li and D. Yeung, "Application-Level Correctness and its Impact on Fault Tolerance," in *Proceedings of the 2007 IEEE 13th International Symposium on High-Performance Computer Architecture*, pp. 181–192, February 2007.

[14] X. Li and D. Yeung, "Exploiting Value Prediction for Fault Tolerance," in *Proceedings of the 3rd Workshop on Dependable Architectures (WDA-III)*, pp. 29–39, November 2008.

[15] P. Dubey, "Recognition, Mining and Synthesis Moves Computers to the Era of Tera," *Technology @ Intel Magazine*, pp. 1–10, February 2005.

[16] U. of California at Berkeley, "The Berkeley Initiative in Soft Computing.".

[17] Y. Jin, "A Definition of Soft Computing.".

[18] S. S. Mukherjee, C. T. Weaver, J. Emer, S. K. Reinhardt, and T. Austin, "A Systematic Methodology to Compute the Architectural Vulnerability Factor for a High-Performance Microprocessor," in *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, pp. 29–40, December 2003.

[19] N. Wang, M. Fertig, and S. J. Patel, "Y-branches: When you come to a fork in the road, take it," in *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, pp. 56–67, September 2003.

[20] J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers, "The Impact of Technology Scaling on Lifetime Reliability," in *Proceedings of the 2004 International Conference on Dependable Systems and Networks*, pp. 177–186, June 2004.

[21] J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers, "Lifetime Reliability: Toward an Architectural Solution," *IEEE Micro*, pp. 70–80, May-June 2005.

[22] S. R. Nassif, "Design for Variability in DSM Technologies," in *Proceedings of the 1st International Symposium on Quality of Electronic Design*, pp. 451–454, March 2000.

[23] G. A. Reis, J. Chang, N. Vachharajani, S. S. Mukherjee, R. Rangan, and D. I. August, "Design and Evaluation of Hybrid Fault-Detection Systems," in *Proceedings of the 32st Annual International Symposium on Computer Architecture*, pp. 148–159, June 2005.

[24] D. Burger, T. Austin, and S. Bennett, "Evaluating future microprocessors: the simplescalar tool set," Tech. Rep. CS-TR-1996-1308, Univ. of Wisconsin - Madison, July 1996.

[25] C. Lee, "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems," in *Proceedings of the 30th annual IEEE/ACM International Symposium on Microarchitecture*, pp. 330–335, December 1997.

[26] J. Pearl, "Probabilistic reasoning in intelligent systems: networks of plausible inference," Morgan Kaufmann Publishers Inc., 1988.

[27] T. Joachims, "Making Large-Scale Support Vector Machine Learning Practical," in *Advances in Kernel Methods: Support Vector Learning*, pp. 169–184, MIT Press, Cambridge, MA, 1998.

[28] V. Kianzad and S. S. Bhattacharyya, "Multiprocessor Clustering for Embedded Systems," in *Proceedings of the 7th International Euro-Par Conference Manchester on Parallel Processing*, (Manchester, United Kingdom), pp. 697–701, August 2001.

[29] V. Kianzad and S. S. Bhattacharyya, "Efficient Techniques for Clustering and Scheduling onto Embedded Multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, pp. 667–680, July 2006.

[30] B. Taskar, M.-F. Wong, P. Abbeel, and D. Koller, "Link Prediction in Relational Data," in *Proceedings of Neural Information Processing Systems*, December 2003.

[31] C. Chang and C. Lin, "LIBSVM : a library for support vector machines.".

[32] T. N. Vijaykumar, I. Pomeranz, and K. Cheng, "Transient-fault recovery using simultaneous multithreading," in *Proceedings of the 29th annual International Symposium on Computer Architecture*, pp. 87–98, May 2002.

[33] J. S. Plank, M. Beck, G. Kingsley, and K. Li, "Libckpt: Transparent checkpointing under unix," in *Proceedings of USENIX Winter1995 Technical Conference*, pp. 213–224, January 1995.

[34] J. S. Plank, Y. Chen, K. Li, M. Beck, and G. Kingsley, "Memory Exclusion: Optimizing the Performance of Checkpointing Systems," *Software – Practice and Experience*, pp. 125–142, February 1999.

[35] K. Wang and M. Franklin, "Highly accurate data value prediction using hybrid predictors," in *Proceedings of the 30th annual IEEE/ACM International Symposium on Microarchitecture*, pp. 281–290, December 1997.

[36] B. Goeman, H. Vandierendonck, and K. de Bosschere, "Differential FCM: Increasing Value Prediction Accuracy by Improving Table Usage Efficiency," in *Proceedings of the 2001 IEEE 7th Annual International Symposium on High-Performance Computer Architecture*, Jan. 2001.

[37] C. Weaver, J. Emer, S. S. Mukherjee, and S. K. Reinhardt, "Techniques to reduce the soft error rate of a high-performance microprocessor," in *Proceedings of the 31st Annual International Symposium on Computer Architecture*, pp. 264–275, June 2004.

[38] N. Wang, A. Mahesri, and S. J. Patel, "Examining ACE Analysis Reliability Estimates Using Fault-Injection," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, pp. 460–469, June 2007.

[39] M. A. Breuer, S. K. Gupta, and T. M. Mak, "Defect and Error Tolerance in the Presence of Massive Numbers of Defects," *IEEE Design and Test Magazine*, pp. 216–227, May-June 2004.

[40] M. A. Breuer, "Multi-media Applications and Imprecise Computation," in *Proceedings of the 8th Euromicro Conference on Digital System Design*, pp. 2–7, September 2005.

[41] J. W.-S. Liu, W.-K. Shih, K.-J. Lin, R. Bettati, and J.-Y. Chung, "Imprecise Computations," *Proceedings of the IEEE*, January 1994.

[42] K. V. Palem, "Energy Aware Algorithm Design via Probabilistic Computing: From Algorithms and Models to Moore's Law and Novemberl (Semiconductor) Devices," in *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, pp. 113–116, October 2003.

[43] K. V. Palem, "Energy Aware Computing Through Probabilistic Switching: A Study of Limits," *IEEE Transactions on Computers*, pp. 1123–1137, September 2005.

[44] C. Alvarez and M. Valero, "A Fast, Low-Power Floating Point Unit for Multimedia," in *2nd Workshop on Application Specific Processors*, pp. 17–24, January 2003.

[45] C. Alvarez, J. Corbal, and M. Valero, "Fuzzy Memoization for Floating-Point Multimedia Applications," *IEEE Transactions on Computers*, pp. 922–927, July 2005.

[46] G. P. Saggese, A. Vetteth, Z. Kalbarczyk, and R. Iyer, "Microprocessor Sensitivity to Failures: Control vs. Execution and Combinational vs. Sequential Logic," in *Proceedings of the 2005 International Conference on Dependable Systems and Networks*, pp. 760–769, June 2005.

[47] N. Wang, J. Quek, T. M. Rafacz, and S. Patel, "Characterizing the effects of transient faults on a high-performance processor pipeline," in *Proceedings*

of the 2004 International Conference on Dependable Systems and Networks, pp. 61–72, June 2004.

[48] S. Kim and A. K. Somani, "Soft Error Sensitivity Characterization for Microprocessor Dependability Enhancement Strategy," in *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pp. 416–425, September 2002.

[49] E. W. Czeck and D. P. Siewiorek, "Effects of Transient Gate-Level Faults on Program Behavior," in *Proceedings of the 1990 International Symposium on Fault Tolerant Computing*, pp. 236–243, June 1990.

[50] P.Shivakumar, M.Kistler, S.W.Keckler, D.Burger, and L.Alvisi, "Modeling the effect of technology trends on the soft error rate of combinatorial logic," in *Proceedings of the International Conference on Dependable Systems and Networks*, pp. 389–398, June 2002.

[51] J. S. Kim, C. Nicopoulos, N. Vijaykrishnan, Y. Xie, and E. Lattanzi, "A Probabilistic Model for Soft-Error Rate Estimation in Combinational Logic," in *Proceedings of the International Workshop on Probabilistic Analysis Techniques for Real-time and Embedded Systems*, (Italy), September 2004.

[52] T. M. Austin, "DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design," in *Proceedings of the 32nd annual IEEE/ACM International Symposium on Microarchitecture*, pp. 196–207, November 1999.

[53] E. Rotenberg, "AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors," in *Proceedings of the 29th International Symposium on Fault-Tolerance Computing Systems*, pp. 84–91, June 1999.

[54] K. Sundaramoorthy and Z. Purser, "Slipstream processors: Improving both Performance and Fault Tolerance," in *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 257–268, Novembermber 2000.

[55] M. Gomaa, C. Scarbrough, T. N. Vijaykumar, and I. Pomeranz, "Transient-fault recovery for Chip Multiprocessors," in *Proceedings of the 30th annual international symposium on Computer architecture*, pp. 76–83, June 2003.

[56] K. M. Chandy and L. Lamport, "Distributed Snapshots: Determining Global States of Distributed Systems," in *ACM Tranaction on Coomputer Systems*, pp. 63–75, February 1985.

[57] J. L. Kim and T. Park, "An Efficient Protocol for Checkpointing Recovery in Distributed Systems," in *ACM Tranaction on Parallel Distributed Systems*, pp. 955–960, 1993.

[58] R. E. Ahmed, R. C. Frazier, and P. N. Marinos, "Cache Aided Rollback Error Recovery (CARER) Algorithms for Shared-Memory Multiprocessor Systems," in *Proceedings of the 20th International Symposium on Fault Tolerant Computing*, pp. 82–88, June 1990.

[59] M. Gomaa and T. N. Vijaykumar, "Opportunistic Transient-Fault Detection," in *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pp. 172–183, June 2005.