ABSTRACT

Title of dissertation: HARNESSING CHECKER HIERARCHY FOR RELIABLE MICROPROCESSORS

Joonhyuk Yoo
Doctor of Philosophy, 2007

Dissertation directed by: Professor Manoj Franklin
Department of Electrical and
Computer Engineering

Microprocessors are becoming increasingly susceptible to soft errors due to the current trends of semiconductor technology scaling. Traditional fault-tolerant multi-threading architectures provide good fault tolerance by re-executing all the computations. However, such a full re-execution significantly increases the demand on the processor resources, resulting in severe performance degradation.

To address this problem, this dissertation presents Active Verification Management (AVM) approaches that utilize a checker hierarchy to increase its performance with a minimal effect on the overall reliability. Based on a simplified queueing model, AVM employs a filter checker which prioritizes the verification candidates so as to selectively do verification. This dissertation proposes three filter checkers—based on (1) result usage, (2) result bitwidth, and (3) result anomaly—that exploit correctness-criticality metrics and anomaly speculation.

Binary Correctness Criticality (BCC) and Likelihood of Correctness Criticality (LoCC) are metrics that quantify whether an instruction is important for reliability

or how likely an instruction is correctness-critical, respectively. Based on the BCC, a result-usage-based filter checker mitigates the verification workload by bypassing instructions that are unnecessary for correct execution. Computing the LoCC is accomplished by exploiting information redundancy of compressing computationally useful data bits. Numerical significance hints let the result-bitwidth-based filter checker guide a verification priority effectively before the re-execution process starts. Extensive measurements prove that the LoCC yields quite a wide distribution of values, indicating that it has the potential to differentiate diverse degrees of correctness-criticality.

A result-anomaly-based filter checker exploits a value similarity property, which is defined by a frequent occurrence of partially identical values. Based on the biased distribution of similarity distance measure, this dissertation further investigates another application to exploit similar values for soft error tolerance with anomaly speculation. Extensive measurements show that the majority of instructions produce values that are different from the previous result value only in a few bits.

Experimental results show that the proposed schemes accelerate the processor to be 180% faster than traditional fully-fault-tolerant processor, with a minimal impact on the overall soft error rate. With no AVM, congestion at the checker badly affects performance, to the tune of 57%, when compared to that of a non-fault-tolerant processor. With good marking by AVM, the performance of a reliable processor approaches that of a processor with no verification. These results explain that the proposed AVM has the potential to solve the verification congestion problem when perfect fault coverage is not needed.

HARNESSING CHECKER HIERARCHY FOR RELIABLE MICROPROCESSORS

by

Joonhyuk Yoo

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
2007

Advisory Committee:
Professor Manoj Franklin, Chair/Advisor
Professor Peter Petrov
Professor Gang Qu
Professor Charles Silio
Professor Eun-Suk Seo

Acknowledgments

Many people generously contributed their love, prayer, time, and support toward the completion of this dissertation. It is my great honor to know them who made this dissertation possible and my life at the University of Maryland joyful. First and foremost, I would like to thank my advisor, Dr. Manoj Franklin for giving me an invaluable opportunity to work with him. It has been always pleasure with me to discuss problems and findings with him over the past three years. His devotion to research and advising has inspired me in the way I should go as a role model in my academic life. Without his instruction and encouragement, my graduate study could neither be enjoyable nor a meaningful part of my life.

I would like to acknowledge the members of my dissertation committee, Dr. Peter Petrov, Dr. Gang Qu, and Dr. Charles Silio, for agreeing to serve on my committee and for sparing their time to review the manuscript. I am also grateful to Dr. Andre Tits and Dr. Gun-A Yoon for their counseling on changing my advisor and on moving forward when I was in despair three years ago. Especially, I would like to thank Dr. Eun-Suk Seo for giving me an opportunity to participate in the development of CDAQ software for NASA's CREAM flight project launched in Antarctica. Due to her encouraging words and financial support, I was able to gain invaluable experience, improve my confidence, and everything turned out all right while working with her.

I specially would like to thank my mom and sisters, who have always been my biggest supporters throughout my entire life, for their endless prayer and love. I

am also grateful to my father-in-law, mother-in-law, brothers-in-law, and sisters-in-law for their enormous encouragement and consistent support. Without their love, it would have been impossible for me to go through every step and arrive at this moment.

During my stay in Maryland for the past seven years, I was so lucky to have many good people around me. I would like to thank my lab-mates in the System and Computer Architecture Laboratory of the University of Maryland for their help and discussion, and sharing ideas: Seungryul, Dongkeun, Hameed, Wanli, Xuanhua, Brinda, Rania, Mike, Sean, Kursad, Ohm, Jaeyong, Inseok, and all the other members. I also would like to acknowledge the Korean graduate students at Maryland and their family for making my life at the University of Maryland full of fun: Seungjong, Woochul, Donghoon, Soobum, Dongik, Younggu, Kyechong, Jusub, Chanho, Eunmi, Hojin, Seokjin, Kwangsik, Keunmin, Sangchul, Jookyung, Kyungil, Hyunmo, Kyungnam, Jongsu, Jiwhang, Yooah, Seungjoon, Minho, Minkyung, Jisun, Ilchul, Jiksoo and many others. In addition, I want to thank Anna and Vinay for sharing good memory as my office-mates. I also want to thank Sonny, Youngsoo, Prashant, Opher, Moohyun, Jihye, Larry, Peter, and other Cosmic Ray Physics Group members for their help and collaboration. I also thank Sangmin, Daesik, Yeonjin, Yoon, Jongso, Hyosang, Hyunchool, and all other Postechians in Maryland. My old friends over twenty years in Korea, Moongab and Euiyoon have always been supportive on me. In particular, I would like to thank Daehyun, Taeheum, Hyejung, Hyunwook, and Hyunmi for their sharing prayer and encouraging me to keep moving forward.

I specially would like to thank my church family for standing by me and praying to God for me and my family in my suffering: Father Valentine Han, Mother Teresa Han, Sister Young Yim, and all other brothers and sisters in the Holy Cross Korean Episcopal Church. With their warm love and prayer, I was able to find a new life in Jesus Christ and finish my graduate study.

Most significantly, I owe my deepest thanks to my dear Eunsook, Changmin, and Jimin. I am so thankful to God for being with them and for giving us such lovely kids, Changmin and Jimin, who have always been the spring of happiness and joyfulness in my family. Eunsook has always been my best friend, my companion in Christ Jesus, my trustful fellow counselor as well as my sweet wife. Any word cannot express how much I owe my gratitude to her and my kids.

It is impossible to remember all, and I apologize to those I have inadvertently left out.

Lastly, I thank God for giving me His grace to know Him during my Ph.D. study, showing me His endless love, and being always with me even though I walk through the valley of the shadow of death. Whenever anxiety was great within me, His consolation always brought joy to my soul. When there seemed to be no way in my study, God prepared a way for me. When trials seemed impossible and I could not face the day, the Lord extended His helping hand and made for me a way. I praise the Lord what He has done in my life.

Table of Contents

List of Figures

List of Abbreviations

λ	Average arrival rate
μ	Average departure rate
ρ	Utilization factor
ACE	Architecturally Correct Execution
AR-SMT	Active-Stream/Redundant-Stream Simultaneous Multi-Threading
AVF	Architectural Vulnerability Factor
AVM	Active Verification Management
BCC	Binary Correctness Criticality
CAP	Congestion Avoidance Policy
CC	Correctness Critical
CMP	Chip Multi-Processor
CNC	Correctness Non-Critical
CNI	Correctness Non-criticality Indicator
CPI	Cycles Per Instruction
DIVA	Dynamic Implementation Verification Architecture
DoU	Degree of Usage
FP	Floating-Point
ILP	Instruction-Level Parallelism
INT	Integer
IPC	Instructions Per Cycle
LCNC	Likely Correctness Non-Critical
LoCC	Likelihood of Correctness Criticality
LSP	Least Significant Partition
LZC	Leading Zero Counter
MITF	Mean Instructions To Failures
MSB	Most Significant Byte
MSP	Most Significant Partition
NSI	Numerical Significance Information
P-ALU	Partitioned ALU
RMT	Redundant Multi-Threading
SER	Soft Error Rate
SEU	Single Event Upset
SMT	Simultaneous Multi-Threading
SoR	Square of Replication
TLP	Thread-Level Parallelism
VCC	Value-based Correctness Criticality
VCD	Value Compression Decoder
VCE	Value Compression Encoder
VI	Value Identification
VR	Value Replication
VVF	Value Vulnerability Factor

Chapter 1

Introduction

1.1 New Challenges for Reliability in Microprocessors

Current technology trends pose new challenges for reliability in microprocessors. Microprocessor performance has been doubling every year and a half for the past three decades, in part due to semiconductor technology scaling, and in part due to innovations in computer architecture and accompanying software. Semiconductor technology scaling has resulted in larger numbers of smaller and faster transistors and is likely to provide billion-transistor integrated circuits in the near future. This comes, however, with a reduction in the critical charge required to maintain proper device state. Such denser designs are more susceptible to permanent and transient faults due to increased design complexity and reduced signal integrity [40, 42, 43, 67].

There are three forces at work that introduce new reliability challenges in VLSI fabrication technologies [4, 5, 40]. Finer feature sizes result in an increased likelihood of noise-related faults, interference from natural radiation sources, and huge verification burdens brought on by increasingly complex designs.

The first challenge is due to noise-related faults. They are the result of electrical disturbances in the logic values in circuits and wires. As processor feature size shrinks, interconnects become increasingly vulnerable to noise induced by other

wires, which is called crosstalk. This effect is due to the increased capacitance and inductance caused by densely packed wires. At the same time, designs are using lower supply voltage levels to decrease power dissipation, resulting in even more susceptibility to noise as voltage margins are decreased.

Secondly, there are a number of radiation sources in nature that can affect the operation of electronic circuits. The two most prevalent radiation sources are cosmic rays and alpha particles. Cosmic rays arrive from space. While most of these rays are filtered out by the atmosphere, some occasionally reach the surface of the earth, especially at higher altitudes. Alpha particles are created when atomic impurities present in materials decay. When these energetic particles strike a very small transistor, they can deposit or remove sufficient charge to temporarily turn the device on or off, possibly creating a logic error. Energetic particles have been a problem for DRAM designs since the late 1970's when DRAM capacitors became sufficiently small to be affected by energetic particles. It is difficult to shield against natural radiation sources. Cosmic rays that reach the surface of the earth have sufficiently high momentum that they can only be stopped with thick and dense materials. Alpha particles can be stopped with thin shields, but any effective shield would have to be free of atomic impurities; otherwise the shield itself would be an additional source of natural radiation. Furthermore, the shielding approach is not cost effective for most system designs. As a result, designers are likely to be forced to adopt fault-tolerant design solutions to protect against natural radiation interference.

Finally, the third challenge to reliability arises from increased design com-

plexity. As designs becomes increasingly complex, their functionality and electrical verification, under all possible combinations that work reliably in varied and occasionally adverse operating conditions, becomes virtually impossible. At the moment, chip vendors spend considerable resources—mostly 80% of design and fabrication time—to verify the correct operation of parts and to avoid reliability hazards. There is no shortage of testimonials from industry leaders warning that increasing complexity is perhaps the most pressing problem facing future microprocessor designs. Without improved verification techniques, future designs will likely be more costly, take longer to design, and include more undetected design errors.

As a result, the degraded reliability is an important challenge in future generations of microprocessor design, and there is an increasing need for fault-tolerant microarchitectures. Section 1.2 further describes soft error impact and its evidences.

1.2 Soft Error Impact and Evidences

Single Event Upsets (SEUs) arise when energetic particles, such as neutron particles from cosmic rays and alpha particles from packaging material, generate electron-hole pairs as they pass through a semiconductor device. Transistor source nodes and diffusion nodes can collect these charges. A sufficient amount of accumulated charge can invert the state of a logic device such as a latch, an SRAM cell, or a gate, thereby introducing a logical fault into the circuit's operation. Because this type of faults do not reflect a permanent device malfunction, they are called *soft* or *transient* [45]. Faults that arise from a permanent device malfunction are called

hard or *permanent* [10, 56, 61].

A device's rate of errors caused by SEUs depends on both the particle flux it encounters and its circuit characteristics. The particle flux depends on the environment. In Denver, Colorado, for example, at an altitude of 1.5 km, the neutron flux from cosmic rays is there three to five times higher than the flux at sea level [84]. Device circuit parameters that influence the error rate include the amount of charge stored, the vulnerable cross-section area, and the charge collection efficiency. As feature sizes shrink, the smaller amount of charge per device makes a particle strike more likely to cause an error, but the reduced cross section makes a strike on any given device less likely. These effects roughly cancel each other for latches and SRAM cells. Thus, the error rate per latch or SRAM bit at a specific altitude will remain roughly constant or decrease slightly for the next several technology generations. However, the chip error rate will grow in direct proportion to the number of bits on the chip [45]. Thus, while Moore's law gives us exponential transistor count increases, this bounty comes at the cost of exponential error rate increases for unprotected chips.

Soft errors caused by cosmic rays are already making an impact in industry. Sun Microsystems acknowledged in 2000 that cosmic ray strikes on unprotected cache memories had caused random crashes at major customer sites in its flagship enterprise server line, losing a major customer to IBM as a result of this episode [6]. Numerous incidents of cosmic ray strikes had been reported after studying the error logs of several large computer systems [46]. The fear of cosmic ray strikes prompted Fujitsu to protect 80% of the 200,000 latches in its recent Sparc processor with some

form of error detection [2].

Various techniques exist to deal with such faults, from special radiation-hardened circuit designs [16], to localized error detection and correction such as parity and ECC [2], to architectural redundancy [4, 54]. However, all these approaches introduce a significant penalty in performance, power, die size, and design time. Consequently, designers must carefully weigh the benefits of these techniques against their cost. Although a microprocessor with inadequate protection from transient faults might prove useless because of its unreliability, excessive protection can make the resulting product uncompetitive in cost, performance, or power consumption. Currently, the frequency of transient faults is low—typically less than one fault per year per thousand computers—making fault tolerance not so attractive for general purpose computers. However, the future microprocessors will be more prone to transient faults due to their smaller feature sizes and higher frequencies. In the near future, even low-end personal computing systems may need support for concurrent fault detection and recovery.

1.3 Motivation

The primary objective of this research is to investigate techniques for incorporating fault tolerance without significant redundant hardware and without loss in performance. Fault tolerance in itself is a research field with a long history. However, fault tolerance in microprocessors has recently got a fresh look from both the industry and the academia for the above stated reasons. A number of novel fault tol-

erance techniques, facilitated by other innovations in microarchitecture, have been proposed recently.

Traditional dual-modular redundant and triple-modular redundant fault tolerance techniques involve replication of the hardware units, and executing multiple copies of the software on these hardware units. The results produced by the multiple hardware units are compared with the help of comparators and voters, which determine the correct result and identify the faulty units, if any. Such hardware-redundant schemes have not been popular for commercial processors because of their huge hardware overhead.

Several researchers have proposed a host of concurrent system-level fault tolerance techniques using watchdog timers, watchdog processors [7, 39, 58, 64, 71], or re-execution processors [4, 27, 44, 48, 50, 51, 54, 68, 72, 83]. In watchdog techniques, additional information on the program control flow is stored in the program by means of signatures or checksums. At run time, watchdog processors recalculate the signatures based on run-time control flow, and compare them against the compile-time calculated signatures. They, however, require modifications to the instruction set architecture, and raise binary compatibility issues. Moreover, while watchdog techniques are just for monitoring control flow errors, re-execution techniques usually include an error recovery mechanism to detect and correct errors in control flow as well as data values generated.

Redundant Multi-Threading (RMT) [44, 51, 54, 68, 72] and Dynamic Implementation Verification Architecture (DIVA) [4, 5] are recently proposed re-execution approaches for concurrent error detection and recovery. RMT techniques provide

fault detection by executing an instruction stream redundantly in separate thread contexts and comparing execution results across the threads. For example, the AR-SMT proposal [54]—the first work of RMT—assumes a baseline SMT processor and enhances the front end of the SMT pipeline to replicate the fetched instruction stream into two separate thread contexts. Both contexts then execute independently and store their results in a reorder buffer. The commit stage is enhanced to compare instruction outcomes and check for inconsistencies. Any such inconsistencies are used to identify transient errors in the execution pipeline. In DIVA-like architectures, the computations done by an out-of-order processor core are verified by a simple in-order checker processor. These full re-execution schemes redundantly execute each instruction, thereby significantly increasing the demand on the processor resources and resulting in severe performance degradation [30, 35, 75, 77, 80].

Most of the previous studies have focused mainly on achieving reliability, without paying much attention on the performance overhead incurred by re-execution. Consider the DIVA proposal given in [4, 5]. This approach uses a simple in-order *checker* processor, as depicted in Figure 1.1 (a). The checker dynamically checks the computations of the regular (complex) processor. The regular processor core supplies to the checker information concerning the executed instructions, such as the input values and memory addresses referenced. The checker processor uses the control and data dependency resolutions done by the regular processor. Therefore, it is possible to trivially parallelize the checking activities. The checker processor only needs to verify each instruction independently, by executing with the precomputed inputs and comparing its output value to the output value obtained by the regular

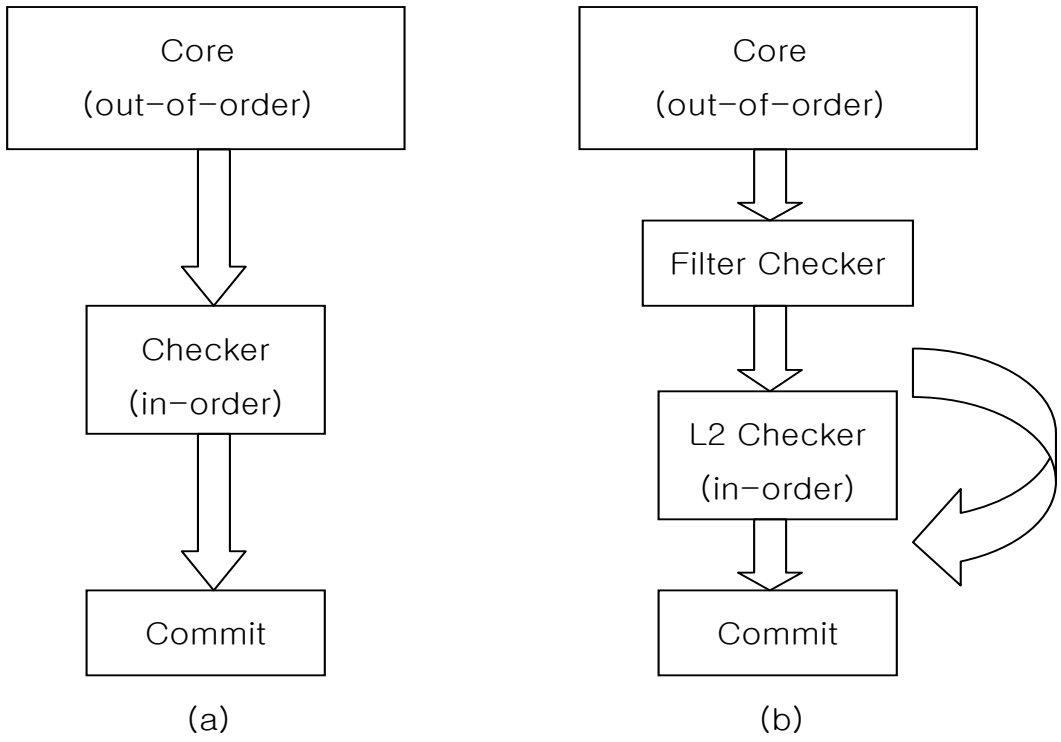


Figure 1.1: Dynamic Verification Architecture: (a) Traditional DIVA, (b) Checker Hierarchy.

processor core. Once each instruction is verified in this manner, then by induction the entire instruction stream is verified. Thus, this arrangement covers design errors as well as transient faults, with only a small amount of additional hardware.

However, dynamic verification with a simple and slow processor core does have one limitation: the checker processor may impact the overall performance, making it inviable for commercial processors. In other words, the throughput of the pair of processor cores would be limited by the checker processor, if it is neither fast nor wide enough to match the main processor [19]. Even though the checker processor does not have to deal with control dependencies and data dependencies, it may encounter some structural hazards because of design decisions geared to reduce hardware cost or power consumption. The hardware resource limitation can increase the checker's latency and reduce its verification bandwidth. Because the checker processor slows the progress of the main processor, the checker becomes a performance bottleneck. We conducted some quantitative analysis to evaluate the impact of checker's congestion on the overall performance. These results, reported in Section 4.3.1, confirm our hypothesis that the performance of a dynamic verification processor is highly dependent on the bandwidth of the checker's pipeline.

In order to be readily acceptable by processor manufacturers and customers, fault tolerance schemes must satisfy [26]:

(1) Low hardware overhead.

(2) Low performance overhead.

(3) Good fault coverage.

The above stated goals are coincident with the primary objective of this study,

which is to investigate techniques for incorporating fault tolerance without significant redundant hardware, without much loss in performance, and with a minimal impact on overall reliability. Therefore, this dissertation proposes a new *Active Verification Management (AVM)* approach with a filter checker to prevent the checker from becoming a performance bottleneck (See Figure 1.1 (b)) [80].

1.4 Contributions

This dissertation makes the following contributions.

- **Sensitivity Analysis of Verification Congestion:** Even though the checker processor has no control dependencies and data dependencies, it may have some structural hazards to reduce hardware cost or power consumption. The checker's congestion can be caused by increased latency or reduced bandwidth due to hardware resource limitation. When the throughput of the core processor exceeds the available bandwidth of the checker processor, the checker processor cannot match the retirement bandwidth of the core processor. This situation is called by *verification congestion*, which will be modeled as a simplified queueing model. When the congestion happens, the performance gets severely degraded and the role of the checker processor becomes important to avoid congestion effectively in dynamic verification. This research will first investigate the impact of checker's congestion on the overall performance, especially when DIVA is employed.

- **Qualitative Analysis of Active Verification Management:** This thesis proposes an Active Verification Management (AVM) approach to prevent the checker from becoming a performance bottleneck. The problem with the current dynamic verification is that it has no congestion management and just stalls until the congestion is resolved. A possible solution for this problem is the Active Verification Management (AVM) concept proposed in our study. AVM is defined by a group of verification management mechanisms to support congestion avoidance in reliable processors. There are two approaches of handling congestion. One is a reactive way to play after the checker is overloaded. Another is a proactive way to play before the checker is overloaded. Several congestion avoidance policies will be discussed. The goal of AVM is to reduce overloaded verification in the checker with a congestion avoidance policy and to achieve minimal performance degradation incurred by congestion.

- **Filter Checker Design and Its Quantitative Analysis:** Before an instruction is verified at the checker processor, a filter checker marks a Correctness Non-criticality Indicator (CNI) bit to indicate how likely its result is to be unimportant for reliability. AVM uses the CNI information to realize a congestion avoidance policy. Both reactive and proactive congestion avoidance policies are proposed to mitigate the performance degradation caused by the checker's congestion. Based on a simplified queueing model, we evaluate the proposed AVM analytically. Our experimental results show that AVM has the potential to solve the verification congestion problem when perfect

fault coverage is not needed. With no AVM, congestion at the checker badly affects performance, to the tune of 57%, when compared to that of a non-fault-tolerant processor. With good marking by AVM, the performance of a reliable processor approaches 95% of that of a processor with no verification. Although instructions can be skipped on a random basis, such an approach reduces the fault coverage. A filter checker with a marking policy correlated with the correctness non-criticality metric, on the other hand, significantly reduces the soft error rate. Finally, we also present results showing the trade-off between performance and reliability.

- **Measuring Correctness Criticality Exploiting Narrow Values:** To address the verification congestion problem, this thesis introduces a correctness criticality based filter checker, which prioritizes the verification candidates so as to selectively do verification. Binary Correctness Criticality (BCC) and Likelihood of Correctness Criticality (LoCC) are metrics that quantify whether an instruction is important for reliability or how likely an instruction is correctness-critical, respectively. A likelihood of correctness criticality is computed by a value vulnerability factor, which is defined by the numerically significant bit-width used to compute a result. The proposed technique is accomplished by exploiting information redundancy of compressing computationally useful data bits. Based on the likelihood of correctness criticality test, the filter checker mitigates the verification workload by bypassing instructions that are unimportant for correct execution. Extensive measurements prove

that the LoCC metric yields quite a wide distribution of values, indicating that it has the potential to differentiate diverse degrees of correctness criticality. Experimental results show that the proposed scheme accelerates a traditional fully-fault-tolerant processor by 1.7 times, while it reduces the soft error rate to 18% of that of a non-fault-tolerant processor.

- **Anomaly Speculation via Value Similarity Prediction:** This dissertation presents a pro-active verification management approach to mitigate the verification workload to increase its performance with a minimal effect on overall reliability. An anomaly-speculation-based filter checker is proposed to guide a verification priority before the re-execution process starts. This technique is accomplished by exploiting a value similarity property, which is defined by a frequent occurrence of partially identical values. Based on the biased distribution of similarity distance measure, this thesis investigates further application to exploit similar values for soft error tolerance with anomaly speculation. Extensive measurements prove that the majority of instructions produce values which are different from the previous result value only in a few bits. Experimental results show that the proposed scheme accelerates the processor to be 180% faster than traditional fully-fault-tolerant processor with a minimal impact on overall soft error rate.

1.5 Road Map

The rest of dissertation is organized as follows. Chapter 2 explains the background of this study, which introduces redundancy based fault tolerance, recent microarchitectural techniques for fault tolerance running the same thread in a single chip, and a methodology measuring reliability to evaluate our proposed schemes.

Chapter 3 proposes the basic idea of *Active Verification Management* techniques, models a dynamic verification process into a simplified queueing-theoretical system, and gives some qualitative analysis results. To evaluate its performance and reliability, we use a simulation-based platform. This experimental platform is also presented in this chapter.

In Chapter 4, a result-usage-based filter checker is implemented to identify correctness-non-critical instructions based on the previous qualitative analysis. This chapter also investigates how much the checker's congestion problem impacts the processor performance. Experimental results show that the implemented filter checker increases the overall performance with a minimal effect on the overall reliability.

In Chapter 5, a result-bitwidth-based filter checker is proposed. Previous binary correctness criticality metric, presented in Chapter 4, is extended into a metric based on value-based correctness criticality. To quantify how likely an instruction is important for correct execution, the likelihood of correctness criticality is defined. A methodology is also proposed to compute the metric. The proposed result-bitwidth-based filter checker is accomplished by exploiting information redundancy

of numerically significant bit-width. In other words, computationally useful data bits are compressed by a dynamic significance compression technique.

Chapter 6 presents a result-anomaly-based filter checker using value similarity prediction. This chapter observes a frequent occurrence of partially identical values in the instruction stream and characterizes value similarity property. Based on its biased distribution, similar values are applied for soft error tolerance with anomaly speculation.

Chapter 7 compares the three filter checkers presented in previous chapters, and discusses the impact of the proposed filter checker on reliability and hardware complexity. It also discusses other potential applications of the proposed active verification management technique.

Chapter 8 compares several related work with our work. Finally, Chapter 9 concludes this dissertation and suggests future research directions.

Chapter 2

Background

2.1 Redundancy Techniques for Fault Tolerance

Redundancy is the best technique to mask transient faults. The basic idea behind any fault tolerance scheme is to have some form of redundancy [26, 27]. This redundancy can be in the form of additional hardware (hardware redundancy), additional software (software redundancy), additional information (information redundancy), or multiple execution of code on a single piece of hardware (time redundancy). Of course, combinations of different types of redundancy can also be employed.

Three key design questions for fault tolerance schemes are:

(1) *For which components will the redundant execution mechanism detect faults?* Components that are difficult to verify due to complexity or sheer number of transistors having high susceptibility to transient faults must take part in re-execution.

(2) *Which inputs must be replicated?* Failure to correctly replicate inputs can result in the divergent execution of redundant operations.

(3) *Which outputs must be compared?* Failure to compare critical values compromise fault coverage. On the other hand, needless comparisons increase overhead and complexity without improving fault coverage.

To answer these questions, the concept of the Sphere of Replication (SoR) [51],

which is defined by the logical extent of redundant execution, was introduced. The sphere of replication abstracts both the physical redundancy as well as logical redundancy. All activities and states within the sphere are replicated, either in time or in space. Components within the sphere enjoy fault coverage due to the redundant execution; components outside the sphere do not, and therefore, must be protected via other means, such as information redundancy. Values that cross the boundary of the sphere of replication are the outputs and inputs that require comparison and replication, respectively.

2.1.1 Hardware Redundancy

Hardware redundant fault tolerance techniques involve replication of the hardware units, and executing multiple copies of the software on these hardware units. The results produced by the multiple hardware units are compared with the help of comparators and voters, which determine the correct result and identify the faulty units, if any. Examples of such schemes are dual-modular redundancy and triple-modular redundancy. Hardware redundant schemes have not been popular for commercial processors because of their hardware overhead. It is also known as *space redundancy*.

2.1.2 Information Redundancy

Information redundancy techniques involve adding additional information to the existing information. Examples are (1) Error-Correcting Code (ECC) for cross-

checking the contents of I-cache, D-cache, register file, and main memory, (2) control flow based signatures for cross-checking run-time control flow, and (3) algorithm-based checksums for cross-checking the generated data values.

ECC is based on systematic application of redundancy to information. That is, in a set of all possible combinations of symbols, only a subset of combinations, called code words, are allowed to be valid combinations so that the occurrence of an error most likely changes it into a non-code word. In control flow checking, additional information on program control flow is stored in the program by means of signatures or checksums. At run time, external devices called watchdog timers or watchdog processors re-calculate the signatures based on run-time control flow, and compare them against the compile-time calculated signatures. This technique detects control flow errors that result in the execution unit taking a path that is not present in the Control Flow Graph (CFG) of the executed program. However, control flow errors due to conditional branches taking incorrect decisions are not detected, because the incorrect paths so taken are still contained in the CFG.

2.1.3 Time Redundancy

Time redundancy involves re-executing a piece of code or an operation using the same piece of hardware, and comparing the two sets of results. At the procedure level, the proposed time redundancy methods are: (1) rollback and recovery schemes employing recovery block and check-pointing methods, and (2) N-version programming. At the instruction execution and data transmission level, the meth-

ods that have been proposed are based on instruction re-execution, re-transmission of data, alternating logic, and re-computation of shifted operands. Recently, several software fault tolerance techniques have been proposed [47, 52, 53].

2.1.4 Exploiting Existing Hardware Redundancy

In high performance processors that exploit some form of parallelism, hardware redundancy in some form is already provided for exploiting concurrency. For example, vector processors routinely pipeline each functional unit heavily. VLIW processors even provide multiple copies of the same hardware functional unit with a view to achieve high peak performance. Multiprocessors provide multiple copies of the processing element to exploit parallelism at a coarser level. Any time extra hardware is thrown in for improving the performance, there is a good chance that the utilization of the hardware becomes low. For instance, in superscalar processors, not all stages of the functional unit pipelines may be busy during every clock cycle. This makes it particularly attractive to employ time redundancy by making use of the redundant hardware in superscalar processors. Similarly, in multiprocessors, not all processors may be active all the time, and this redundancy can be used for executing identical copies of programs.

Recent commercial single-chip multithreading processors allow concurrent execution of multiple threads in a single chip by maintaining multiple on-chip hardware contexts. Running multiple threads in a single chip gives an opportunity to implement fault tolerance schemes that exploit existing hardware redundancy. This

architecture effectively utilizes the ever-increasing hardware budget available in a single chip. In addition, this architecture is a cost-effective way of exploiting Thread-Level Parallelism (TLP) because it allows some of the on-chip hardware resources to be shared between concurrently running threads, rather than dedicating them to individual threads.

Depending on the design of the single-chip multithreading processors, the choice of the dedicated and shared hardware resources varies. Two extremes of single-chip multithreading processor design are Chip Multi-Processor (CMP) and Simultaneous Multi-Threading (SMT) processor. SMT allows the execution of multiple threads in a single core by letting fine-grained sharing of most of the processor resources, as well as the L1 cache and the L2 cache between concurrently running threads. The only resources dedicated to each thread are the program counter and additional storage to maintain context information. CMP has multiple processor cores in a single chip. Each core has its own dedicated processor resources, including branch predictor, fetch queue, issue queue, functional unit, memory port, register file, and reorder buffer, to execute a thread. However, multiple cores share the on-chip L1 and/or L2 caches.

SMT can utilize the processor resource more efficiently because SMT allows one thread to use almost all of the shared resources when the other thread(s) cannot fully utilize them. SMT achieves higher per-core throughput by exploiting parallelism between independent threads. However, the increased processor throughput comes at the expense of single-thread performance. Because multiple threads share hardware resources at the same time, individual threads get fewer resources than

what they would have received if they had been running alone.

On the other hand, the multiple cores in a CMP are duplicates of a single core. Since each core in CMP is independent of each other, increasing the number of cores in a chip does not severely increase the complexity of the interconnections within a chip, making it more scalable. CMP may have either heterogeneous cores, with either powerful out-of-order processor core(s) mixed with simple small in-order processor core(s), or homogeneous cores.

Due to these advantages of single-chip multithreading processor design, many CMP and SMT processors are commercially available nowadays. Intel Pentium4 with Hyper-threading is an SMT product. IBM Power4, AMD Athlon64 dual core, Intel Pentium dual core, and Intel Pentium quad core are all CMP products. IBM Power5 architecture has two SMT cores in a single chip, making it a hybrid of both SMT and CMP.

2.2 Running the Same Thread in a Single Chip

An alternative of running multiple threads that several researchers have proposed is to execute the same instructions in multiple contexts [57]. Although it may seem counter-intuitive, there are several potential benefits to such an approach. The first proposal to suggest doing so, Active-stream/Redundant-stream Simultaneous Multi-Threading (AR-SMT), focused on fault detection. By executing an instruction stream twice in separate thread contexts and comparing the execution results across the threads, transient faults in the processing pipeline can be detected. In

other words, if the pipeline hardware flips a bit due to a soft error in a storage cell, the likelihood of the same bit being flipped in the redundant stream is very low. Comparing results across threads will likely detect many such transient faults.

The AR-SMT proposal assumes a baseline SMT processor and enhances the front-end of the SMT pipeline to replicate the fetched instruction stream into two separate thread contexts. Both contexts then execute independently, and store their results in a reorder buffer. The commit stage of the pipeline is further enhanced to compare instruction outcomes, as they are committed, to check for inconsistencies. Any such inconsistencies are used to identify transient faults in the execution pipeline. A similar approach is used in real processor designs that place emphasis on fault detection and fault tolerance. For example, the IBM S/390 G5 processor also performs redundant execution of all instructions, but achieves this by replicating the pipeline hardware on chip and running both pipelines in lock step [63]. Similar system-level designs were available from Compaq's Tandem division. In these designs, two physical processor chips are coupled to run the same threads in a lockstep manner, and faults are detected by comparing the results of the processors to each other.

An interesting observation grew out of this AR-SMT study, say the active and redundant streams end up helping each other execute more efficiently. This cooperative behavior has been exploited by prefetching memory references and resolving branch mispredictions for each other [22, 69, 70] via critical path instructions. One positive side effect of redundant execution is prefetching because both threads are generating the same stream of instruction and data memory references. Whenever

one thread runs ahead of the other, it prefetches useful instructions and data into the processor's caches. This can result in a net speedup, since additional memory-level parallelism is exposed. Another main benefit of redundant execution is early resolution of branch instructions that are hard to predict with conventional approaches to branch prediction. Figure 2.1 illustrates these uses for running the same thread in a single chip. Figure 2.1 (a) shows how a runahead thread can prefetch cache misses and resolve branch misprediction for the main thread, while Figure 2.1 (b) shows how a redundant thread can be used to check the main thread for transient faults.

Both Redundant Multi-Threading (RMT) and Dynamic Implementation Verification Architecture (DIVA) processors, which will be introduced in Sections 2.2.1 and 2.2.2, respectively, build on the AR-SMT concept and pursue potential benefits of fault detection as well as pre-execution.

2.2.1 Redundant Multi-Threading

Redundant Multi-Threading (RMT) uses two threads running the same program on an SMT processor [29, 44, 51, 54, 68, 72]. Program-level time redundancy effectively doubles the execution time of a program because the same program is run twice back-to-back. As the programs are run sequentially there is great performance loss. SMT is identified to provide time redundancy using re-execution as well as providing a performance boost to fault tolerant architectures by maximizing ILP. SMT is a technique that permits multiple independent threads to issue multiple

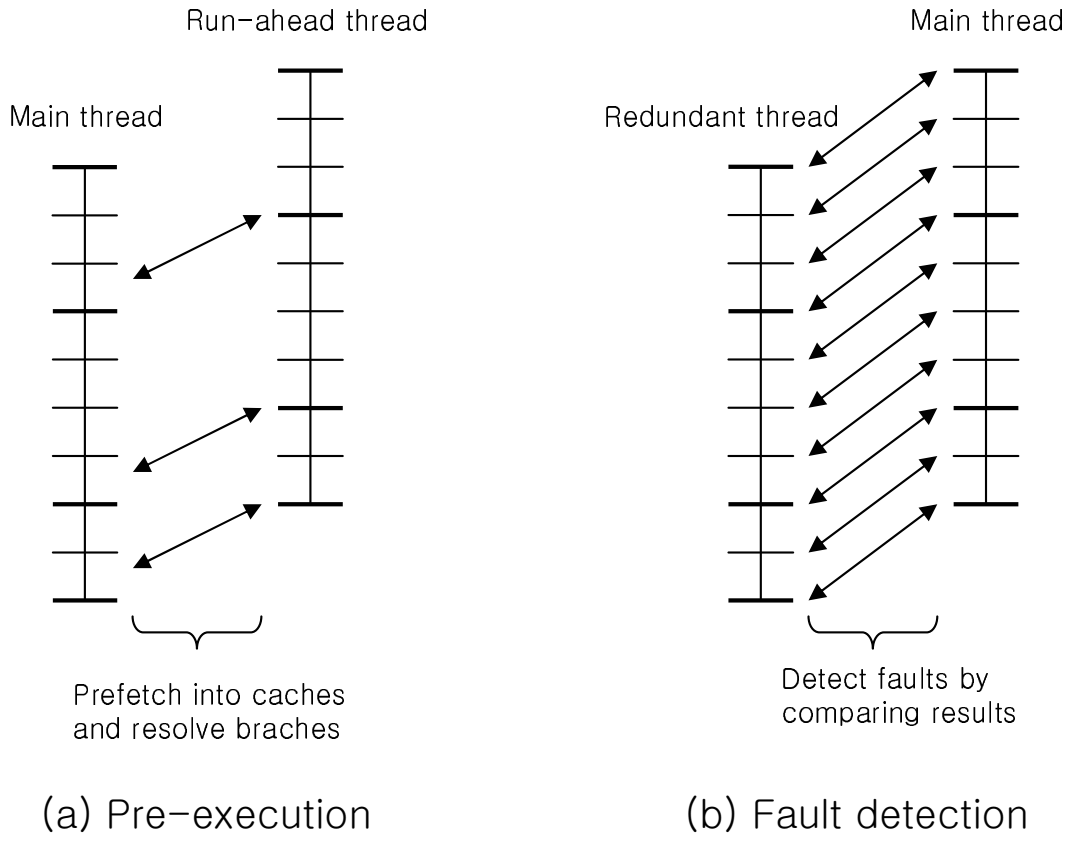


Figure 2.1: Comparison of Pre-execution and Fault Detection: (a) pre-execution, (b) fault Detection [57].

instructions each cycle to a superscalar processor's functional units. This dynamic sharing of the functional units allows SMT to substantially increase throughput, attacking the two major impediments to processor utilization —long latencies and limited per-thread parallelism.

In Redundant Multi-Threading (RMT), two explicit copies of the same program run concurrently on the same processor resources. The two copies are treated as completely independent programs, each having its own state or program context. Consequently, as with program-level time redundancy, the entire pipeline of the processor is conceptually duplicated, providing broad coverage of the chip. RMT uses SMT's multithreaded execution to replicate an application into two communicating threads, one executing ahead of the other. Comparing the results of two redundant executions is the underlying scheme to detect transient faults in RMT.

RMT being based on the concept of SoR entails that (1) all computation and data within this sphere are replicated such that each thread uses its own copy, (2) data entering the SoR is independently read by the two threads using input replication, (3) data exiting the SoR from the two threads are compared using output comparison, and only one copy of the checked data is stored outside the SoR. Both RMT and DIVA processors encompass the processor core as their Square of Replication (SoR).

2.2.2 Dynamic Implementation Verification Architecture

The DIVA processor also builds on the AR-SMT concept, but instead of using two threads running on an SMT processor, it adds a simple in-order checker processor that dynamically checks the computations of a regular out-of-order processor by re-executing the instruction stream [4, 5, 19, 18, 49]. The main motive in adding a checker processor is to provide dynamic verification as compared to the static verification employed in contemporary designs. The basic idea is to make the execution of the main processor speculative and then verifying the speculative results using a checker. To implement dynamic verification, a microprocessor is constructed using two heterogeneous internal processors that execute the same program. The main processor is responsible for pre-executing the program to create the prediction stream. The prediction stream consists of all executed instructions delivered in program order with their input values and any memory addresses referenced. The checker processor follows the regular processor, verifying the activities of the main processor by re-executing all program computation in its wake. The checker processor is assumed to be correct since its simple design lends itself to easy verification. The speculative stream from the main processor serves to simplify the design of the checker processor and speed its processing. Figure 2.2 illustrates the DIVA proposal.

Pre-execution of the program on the complex regular processor eliminates all the processing hazards, for example branch mispredictions, cache misses, and data dependences, that slow simple processors and necessitate complex microarchitec-

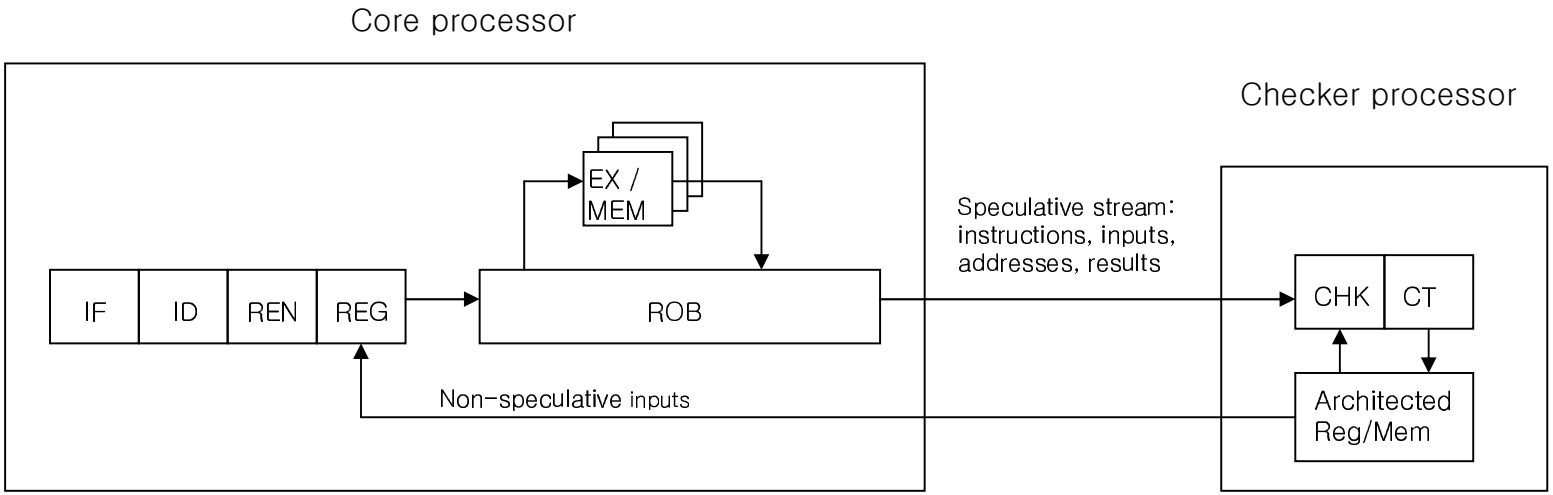


Figure 2.2: Dynamic Implementation Verification Architecture [4].

ture. By removing all dependences in the checker, the code running on the simple processor can be parallelized. The simple processor need only verify each instruction in isolation, by executing with the provided inputs and comparing the output to the provided output. Once each instruction is verified in this manner, then, by induction, the entire instruction stream is also verified. In the event the main processor produces a bad prediction value due to a design error, the checker processor will detect the bad value and flush all internal state from the main processor and restart it after the errant instruction. Once restarted, the main processor will resynchronize with the correct state of the machine as it reads register and memory values from non-speculative storage. The resulting dynamic verification architecture should therefore benefit from a reduced burden of verification, because only the checker needs to be built correctly. Since the simple processor is by definition easy to verify for correctness, it can be trusted to check the operation of the much more complex and design-error-prone runahead processor. Hence, the checker processor will fix any errors in the main processor.

For dynamic verification to be viable, the checker processor must be simple and fast. It must be simple enough to reduce the overall design verification burden, and fast enough to not slow the main processor. A single issue two-stage checker processor is illustrated in Figure 2.3. The checker processor has two modes—the CHECK mode and the RECOVER mode. When the main processor retires an instruction, the checker pipeline receives an instruction with main processor predictions. These predictions include the next PC, instruction, instruction inputs, and addresses referenced for loads and stores. The checker processor ensures the

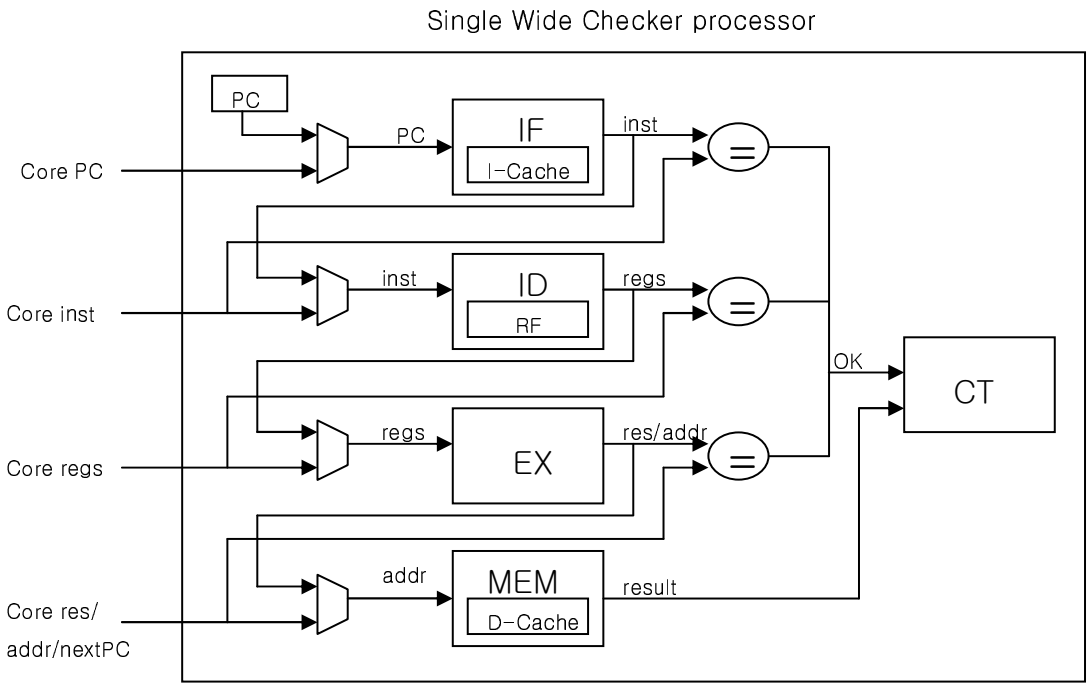


Figure 2.3: Single-wide Checker Processor Pipeline [4].

correctness of each component of this transfer by using four parallel stages, each of which verifies a separate component of the prediction stream. Each parallel stage implements a substep of instruction execution and verifies whether the computed value is identical to that received from the main processor. If each prediction from the core processor is correct, the checker processor is allowed to retire the instruction to non-speculative storage in the commit stage of the checker processor.

In the event any prediction information is found to be incorrect, the checker enters the RECOVER mode. The bad prediction is fixed, the main processor is flushed, and restarted after the errant instruction. The main processor's flush and restart use the existing branch speculation recovery mechanism contained in all modern high-performance pipelines. In the recovery mode, the pipeline is recon-

figured into a serial pipeline, very similar to the classic five-stage pipeline. In this mode, stage computations are sent to the next logical stage in the checker processor pipeline, rather than used simply to verify main predictions.

Unlike the classic five-stage pipeline, only one instruction is allowed to enter the recovery pipeline at a time. As such, the recovery pipeline configuration does not require bypass datapath or complex scheduling logic to detect hazards. Once the instruction has retired, the checker processor reenters normal processing mode and restarts the main processor after the errant instruction. An important aspect of the checker design is that the CHECK mode and the RECOVER mode use the same checking modules, thereby reducing the area cost of the checker and its design complexity.

At first glance, the simple checker processor may seem to be able to easily keep up with the main processor because it exploits the fact that the regular processor has speculatively resolved all control flow and data flow dependences. In fact, however, if the checker processor runs slower or it does not support a wider pipeline than the main processor, the checker becomes the execution bottleneck. That is to say, the throughput of the pair of processors would be limited by the simpler one, resulting in poor performance. In the original DIVA proposal, there are three ways the checker processor can slow the progress of the main processor. First, any contention for the ports between the checker and main processors will lead to main processor stalls. Second, the checker processor pipeline delays the retirement of instructions, forcing the main processor to hold speculative state longer, thus creating back-pressure at retirement. If speculative state resources fill, the core processor decoder will stall as

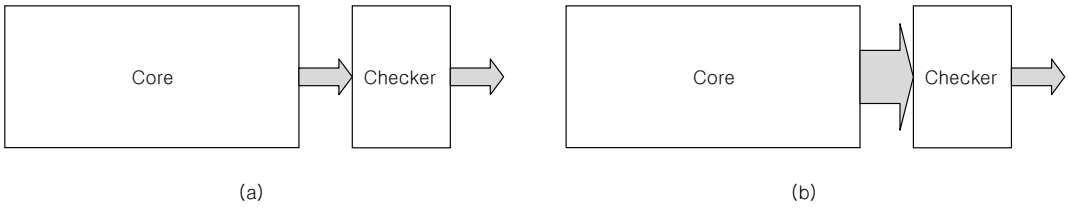


Figure 2.4: Limitation of Dynamic Verification: (a) no performance loss in steady state, (b) performance loss due to checker's congestion.

it will not be able to allocate reorder buffer and load/store queue resources. Finally, checker processor cache misses stall the entire checker pipeline, which again can lead to increased pressure on the main processor speculative state.

As stated above, in the DIVA processor, the checker processor may have a significant impact on the overall performance, making it inviable for commercial processors. Figure 2.4 illustrates the limitation imposed by the checker's congestion. In this dissertation, Section 4.3.1 further analyzes the impact of congestion in the checker processor. Although this quantitative study mainly deals with the performance bottlenecks in the DIVA processor, RMT processors similarly incur a noticeable performance penalty because the main thread shares its resource with the redundant thread.

2.3 Measuring Reliability

2.3.1 Computing a Processor's Soft Error Rate

A processor's Soft Error Rate (SER) is classified into two categories, which are Silent Data Corruption (SDC) and Detected Unrecoverable Error (DUE) [76]. SDC

occurs when an unprotected bit sustains a single bit upset leading to undetected incorrect system behavior. In contrast, a DUE event occurs when an error in a bit is detected, but the system cannot recover from that error. This dissertation mainly focuses on SDC and refers SER to SDC error rate hereafter.

A key aspect of generating SER estimates is that not all faults in a microarchitectural structure affect a program's final outcome. As a result, an estimate based only on raw device fault rates will be pessimistic, leading architects to overdesign their fault-handling features. The probability that a fault in a processor structure will result in a visible error in a program's final output is called that structure's Architectural Vulnerability Factor (AVF) [45]. AVF ranges from 0 to 1. For example, a single bit fault in a branch predictor will not affect the results of any committed instructions, and so the branch predictor's AVF is 0. In contrast, a single bit fault in the Program Counter (PC) will cause the wrong instructions to execute, almost certainly affecting the program's result; hence the PC's AVF is effectively 1.

The overall SER of a microarchitectural structure is the product of its raw fault rate and its AVF. By summing the contributions of all on-chip structures, a processor architect can map the raw fault rate to an overall processor SER and thus determine whether the design meets its SER goals set for the target market. Significantly, this lets the architect examine relative contributions of various structures and identify the most cost-effective areas in which to use fault protection techniques.

To estimate AVFs [9, 45, 82], we use an approach that tracks the subset of processor state bits required for Architecturally Correct Execution (ACE)—any execution that generates results consistent with a system's correct operation as observed

by a user. Any fault in a storage cell that contains one of these bits, called *ACE bits*, will cause a visible error in a program's final output in the absence of error correction techniques. The remaining processor state bits are called *un-ACE bits* because their specific values are unnecessary for architecturally correct execution. A fault that only affects un-ACE bits will not cause an error. The AVF of a single bit storage cell is simply the fraction of time that it holds ACE bits. Assuming that all cells have equal raw fault rates, a structure's AVF is the average of its storage cells or the average fraction of its cells holding ACE bits at any time.

The branch predictor's AVF is thus 0 because all predictor bits are always un-ACE bits. Similarly, all the bits in the PC are always ACE bits, leading to its AVF of 1. The real power of ACE-bit analysis lies in computing the AVFs for structures that hold ACE bits at some times and un-ACE bits at other times. Rather than enumerating for each structure which bits might matter and which might not, the ACE bits are simply determined in the pipeline, the average number of ACE bits in each structure in each cycle is determined, and the ratios of these numbers to the structure's bit capacities are obtained. The average ACE bits can be used for this calculation because fault-inducing particle strikes are randomly and uniformly distributed in a structure.

Using this performance model, the ACE bits in a structure are directly counted. A structure's AVF can also be estimated by counting the ACE bits that flow through the structure and then applying the Little's law, which states that the average capacity of an open system is the product of the bandwidth of individual objects flowing through the system and the average residence time of each object in the system [8].

It is difficult to precisely classify ACE and un-ACE bits over a program's entire execution. Instead, it is assumed conservatively that every bit is an ACE bit unless it can be proved to be un-ACE. Thus, an upper bound of the AVF number is computed by obtaining a conservative estimate of a processor's AVF. Five sources of architectural un-ACE bits has been identified—(1) nop instructions, (2) performance enhancing instructions such as prefetches, (3) predicated-false instructions, (4) dynamically dead instructions, and (5) logically masked instructions. The results of dynamically dead instructions either are never used by any subsequent instruction in a program or are used only by other dynamically dead instructions [15]. Most of bits of such instructions can be classified by un-ACE, except the opcode and some specific bits, when they are stored in processor structures.

With the Little's law, the average number of ACE bits resident in a structure and therefore the structure's AVF can be computed. The Little's law is translated by $N = B \times L$, where N is the average number of bits in a processor structure, B is the average bandwidth of bits per cycle into the structure, and L is the average residence time of an individual bit in the structure. Applying this equation to ACE bits, the average number of ACE bits in a structure is obtained by the product of the average bandwidth of ACE bits going into the structure (B_{ACE}) and the average residence cycles of an ACE bit in the structure (L_{ACE}). Thus, the structure's AVF can be expressed as

$$\frac{B_{ACE} \times L_{ACE}}{\text{Total number of bits in hardware structure}} \tag{2.1}$$

This formulation is particularly useful in the very early stages of an industrial

processor's design cycle when even a performance model may not be available. Alternatively, in many cases, hardware performance counters can be used to compute the bandwidth of ACE bits going into a structure and the average residence cycles of ACE bits, allowing AVF estimation without a performance model.

2.3.2 Computing Mean Instructions To Failure (MITF)

Traditionally, the fault tolerance community has used the terms MTBF (Mean Time Between Failures) and MTTF (Mean Time To Failure) to reason about error rates in processors and systems. These are usually expressed in years. Typically, MTTF corresponds to system uptime and is related to MTBF as follows: $MTBF = MTTF + MTTR$, where MTTR is the mean time to repair. Because MTTF is usually orders of magnitude greater than MTTR, people often use MTBF and MTTF synonymously. Nevertheless, MTTF is a more appropriate term for processor vendors, such as Intel and AMD, because such vendors do not have control over system-level MTTR-related features, which typically reside outside the processor chip.

While MTTF provides a metric for error rates, it does not allow us to reason about the trade-off between error rates and the performance of a processor. The concept of MITF (Mean Instructions To Failure) has been introduced as an approach to reason about this trade-off [76]. MITF tells us how many instructions a processor will commit, on average, between two errors. MITF is related to MTTF as follows:

$$MITF = \frac{\text{number of committed instructions}}{\text{number of errors encountered}} \tag{2.2}$$

$$= \frac{\text{number of committed instructions}}{\frac{\text{total execution time in cycles}}{\text{frequency} \times MTTF}} \quad (2.3)$$

$$= IPC \times \text{frequency} \times MTTF \quad (2.4)$$

For example, a processor running at 2 GHz with an average committed Instructions Per Cycle (IPC) of 2 and MTTF of 10 years would have a MITF of 1.3×10^{18} instructions.

A higher MITF implies a greater amount of work done between errors. Assuming that, within certain bounds, increasing the MITF is desirable, the MITF can be used to reason about the trade-off between performance and reliability. Since $MTTF = \frac{1}{\text{raw error rate} \times AVF}$, we have:

$$MITF = \frac{IPC \times \text{frequency}}{\text{raw error rate} \times AVF} \quad (2.5)$$

$$= \frac{\text{frequency}}{\text{raw error rate}} \times \frac{IPC}{AVF} \quad (2.6)$$

Thus, at a fixed frequency and raw error rate, MITF is proportional to the ratio of IPC to AVF. It can be argued that mechanisms that reduce both the AVF and the IPC may be worthwhile only if they increase the MITF. In order words, if they increase the IPC-to-AVF ratio by reducing the AVF relative to the base case to a greater degree than they reduce the IPC.

Although the MITF can be used to reason about performance versus AVF for incremental changes, we need to be cautious not to misapply it. For example, it could be argued that doubling processor performance while reducing the MTTF by 50% is a reasonable trade-off, as the MITF would remain constant. However, this explanation may not be adequate for customers who see their equipment fail twice as often.

Chapter 3

Active Verification Management Approach

This dissertation proposes an Active Verification Management (AVM) approach to prevent the checker from becoming a performance bottleneck, as illustrated in Figure 1.1 (b) [77]. This chapter presents the basic idea of AVM in Section 3.2. The AVM is abstractly formulated as a simplified queueing model in Section 3.1 and qualitatively analyzed in Section 3.3. Finally, Section 3.4 describes our simulation methodology.

3.1 A Simplified Queueing Model

The dynamic verification process can be viewed as a verification queue that services jobs that are submitted by the main processor core, as shown in Figure 3.1. Completed instructions in the reorder buffer of the main processor are waiting to be verified by the checker. The number of waiting instructions depends on the processing speed and bandwidth of the checker. Therefore, a dynamic verification architecture can be modeled as a simplified queueing system.

The main processor core is assumed to complete instructions at an average rate of λ and the verification queue is assumed to service jobs at an average rate of μ. The utilization factor ρ is defined by $\frac{\lambda}{\mu}$, which can be interpreted by the fraction of time that the verification queue is busy. If $\rho < 1$, then the service rate exceeds

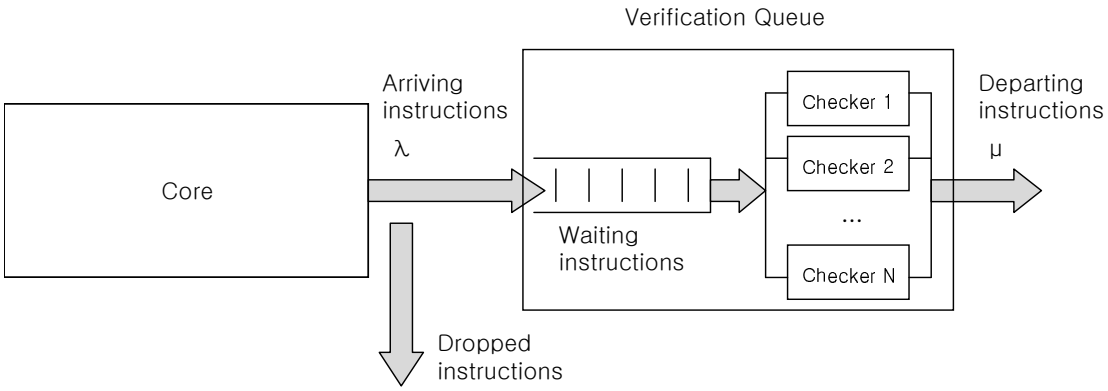


Figure 3.1: A Simplified Model of Dynamic Verification Process.

the arrival rate, and there is no performance degradation. However, if $\rho > 1$, then verification congestion happens and the checker cannot keep up with the retirement bandwidth of the main processor. In steady state, the average number of instructions waiting in the reorder buffer and the average residence time of instructions in the reorder buffer then increases. Therefore, this dissertation presents a novel approach, which controls the verification activity executed at the checker processor, to mitigate the performance degradation caused by the checker's congestion. To illustrate the effectiveness of using AVM, Section 3.3 qualitatively analyzes the simplified queueing model.

3.2 Basic Idea

When the throughput of the main processor exceeds the available bandwidth of the checker processor, the checker processor cannot match the retirement bandwidth of the main processor.

When congestion happens at the checker, the performance is severely affected.

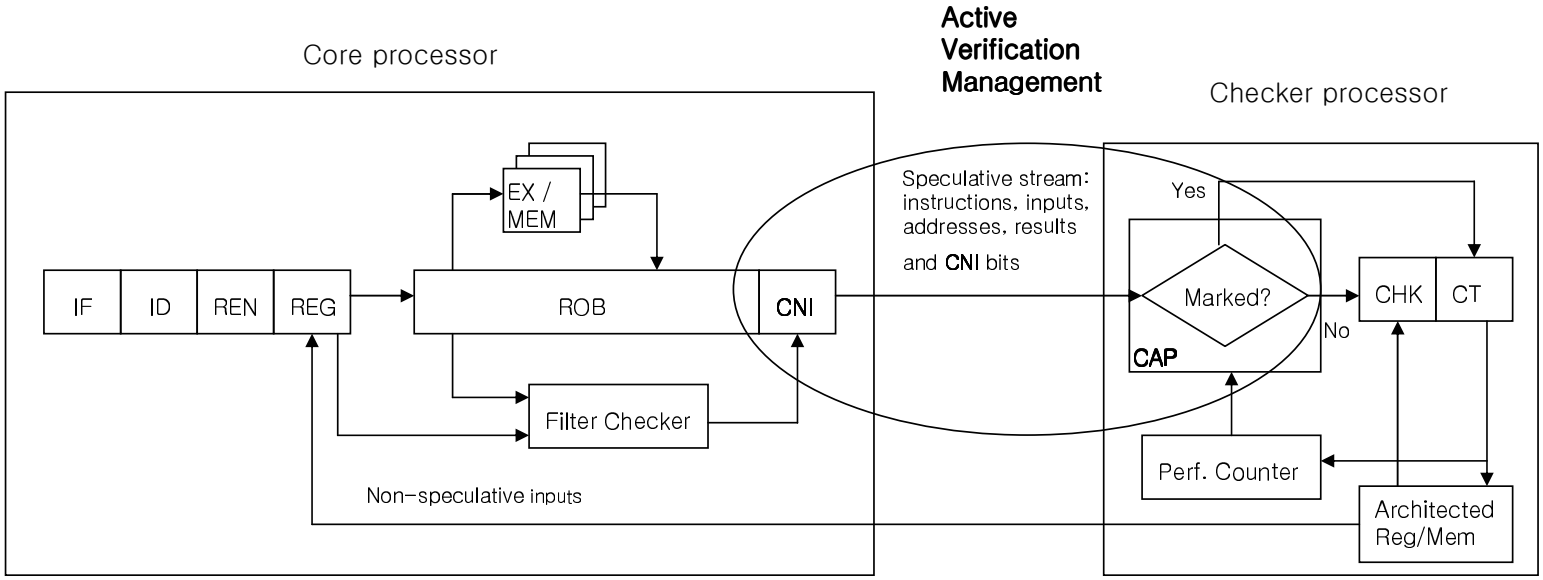


Figure 3.2: Active Verification Management.

The problem with current simple verification schemes is that they have no congestion management and just stall the main processor until the congestion is resolved. We propose *Active Verification Management (AVM)* as a solution to this problem. AVM is defined by a group of verification management mechanisms to support congestion avoidance in fault-tolerant processors. We present two approaches of handling congestion. One is a reactive way to play *after* the checker is overloaded. Another is a proactive way to play *before* the checker is overloaded. Both of these policies are discussed in detail in Section 4.2. The proposed AVM covers design faults as well as soft errors in control logic, processor storage structures such as the register file and reorder buffer, and the functional units. Errors occurring in the checker processor can also be corrected as long as the same error does not occur in both the main and checker processors.

The goal of AVM is to reduce overloaded verification in the checker with a congestion avoidance policy and to achieve minimal performance degradation when congestion occurs. The basic idea is similar to that of a cache hierarchy. Since a small cache is fast and a fast cache is expensive, a cost-effective hierarchical solution can be achieved to obtain both fast speed and large size by using the principle of locality. In our AVM proposal, a flag called *Correctness-Noncriticality Indicator* (CNI) bit is associated with each instruction to indicate if its result is non-critical from the overall correctness point of view. The filter checker implementation to identify correctness-noncritical instructions is described in Section 4.1. Once the CNI bit is updated for an instruction, AVM decides how to deal with the identified instructions by using a congestion avoidance policy (described in Section 4.2). For example, marked

instructions may be directly passed to the commit stage by skipping verification, while un-marked instructions may proceed to the second-level checker for further verification.

3.3 Qualitative Analysis of the AVM Model

Because this study focuses on the case when the checker's congestion happens, let us assume that $\rho > 1$ in the simplified queueing model. We further assume that the performance of reliable processors should be kept the same as that of non-fault-tolerant processors to evaluate the proposed AVM analytically. If the verification queue cannot be allowed to overflow in the proposed AVM model, then the utilization factor should be less than 1. This gives an upper bound on the fault coverage to analyze the effectiveness of various AVM schemes.

Definition 3.1 *The fault coverage is defined by the success probability that the faulty instructions are verified by the checker.*

Lemma 3.1 *The fault coverage of random marking is bounded above by* $\frac{1}{\rho}$.

Proof If AVM randomly skips each instruction with probability p_s, then the utilization factor becomes $(1 - p_s)\rho$. According to the above condition, say with $\rho < 1$, p_s should be chosen such that $p_s > 1 - \frac{1}{\rho}$. The event that any soft error is not verified by the checker is equally probable to the event that the faulty instruction is one of the randomly skipped instructions. Therefore, the success probability that the faulty instructions are verified by the checker is upper-bounded by $\frac{1}{\rho}$. □

Lemma 3.2 *The fault coverage of correlated marking is bounded above by* $(1 - p_{mf})\frac{1}{(1-p_{mc})\rho}$.

Proof Now let us assume that the filter checker marks correct instructions with a probability p_{mc} and faulty instructions with a probability p_{mf}. Each instruction marked by the filter checker can be skipped and can directly proceed to the commit stage, which gives us the utilization factor of $(1-p_{mc})\rho$. Assuming that this is still too high to protect the performance degradation, correlated marking can skip additional instructions un-marked by the filter checker with a probability p_s so that correlated marking can let the utilization factor be less than 1. Then, the new utilization factor becomes $(1 - p_s)(1 - p_{mc})\rho$. Since this should be less than 1, correlated marking should have p_s satisfying $p_s > 1 - \frac{1}{(1-p_{mc})\rho}$. Because the filter checker can mis-speculate the faulty instructions to be correct with a probability p_{mf}, the success probability that the faulty instructions are verified by the checker is upper-bounded by $(1 - p_{mf})\frac{1}{(1-p_{mc})\rho}$. □

Theorem 3.1 *Correlated marking has better fault coverage than random marking with the same performance, if* $p_{mc} > p_{mf}$.

Proof From Lemma 3.1 and Lemma 3.2, the upper bound of correlated marking is superior to the upper bound of random marking because p_{mc} is much greater than p_{mf} for the implemented filter checker. Thus, we have the theorem. □

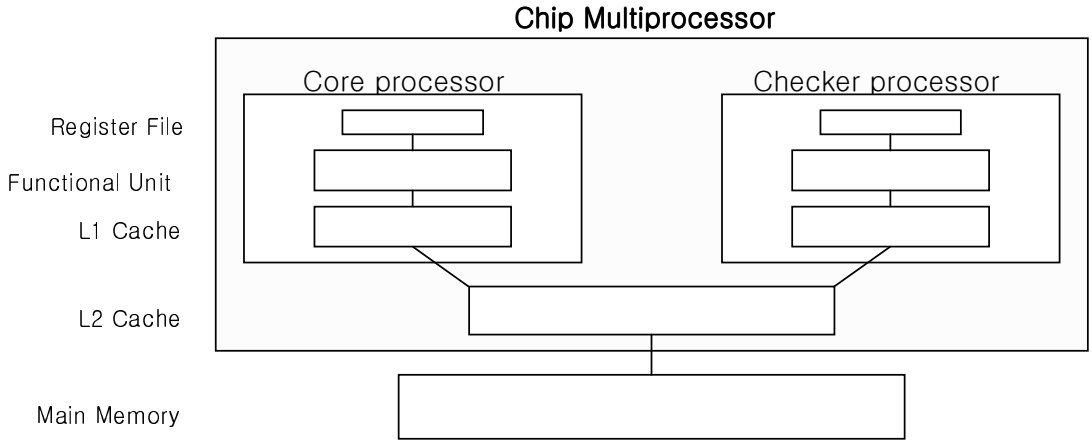


Figure 3.3: Hardware Platform.

3.4 Simulation Methodology

3.4.1 Hardware and Software Platform

Our experiments are performed on a detailed event-driven CMP processor simulator that models the processor pipeline as illustrated in Figure 3.3. The simulator is derived from the M5 Simulator System, an extension of the out-of-order processor model in SimpleScalar [13], and has been used previously to study several techniques. For our evaluation, we model a heterogeneous CMP processor with 4-way issue out-of-order superscalar processor and one simple in-order checker processor. The processor and memory system settings for the baseline complex regular processor are listed in Table 3.1. The baseline checker processor has a 4-issue 2-stage checker pipeline, which is an in-order single CPI machine with its own register file, functional units, and L1 cache of the same type as the regular processor.

Our study is driven by SPEC CPU2000 benchmarks. Table 3.2 lists our bench-

Parameter	Value
Fetch Queue Size	32 instructions
Fetch/Decode/Commit Width	4 instructions
Branch Predictor	4K entry BTB and hybrid predictor
Return Address Stack Size	16 entries
Branch Misprediction Recovery Latency	4 cycles
Physical Register File	128 INT, 128 FP
Issue Width	4 INT, 4 FP
Issue Queue Size	32 INT, 32 FP
Load-Store Queue / Reorder Buffer	64 / 256 entries
INT Functional Units	6 ALU, 2 MUL/DIV
FP Functional Units	4 ALU, 2 MUL/DIV
L1 I-Cache	64K 2-way set-associative, 64B line
L1 D-Cache	64K 2-way set-associative, 64B line, 3 cycles
L2 Cache (Shared)	2M 32K set-associative, 64B line, 10 cycles
Memory Latency	100 cycles

Table 3.1: Baseline Core Processor Hardware Parameters.

marks. We use the pre-compiled alpha binaries, which were built with the highest level of compiler optimization. All of our benchmarks use the reference input set provided by SPEC. We selected simulation regions of our workloads in the following way [21]. First, we used SimPoint [59, 60] to analyze the first 16 billion instructions (or the entire execution, whichever is shorter) of each benchmark, and picked the earliest representative region reported by SimPoint. In our simulations, we fast-forward each benchmark to its representative region. Table 3.2 reports the number of skipped instructions in each benchmark during the fast forwarding. Finally, we turn on detailed simulation, and simulate for 10M on-line instructions executed. Due to the cost of simulation, we are unable to simulate more instructions. However, the regions we simulate are representative thanks to the SimPoint analysis.

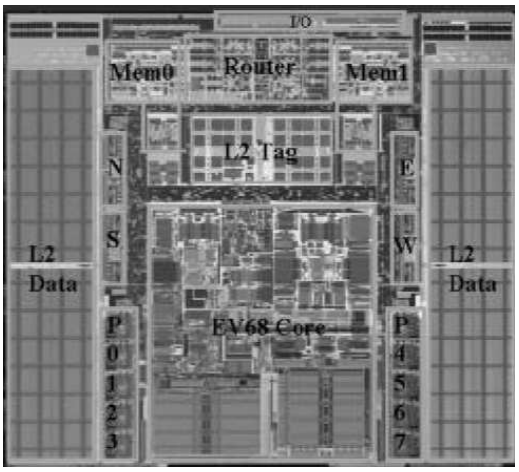
3.4.2 Evaluation

After computing each individual structure's AVFs, they are combined to produce the overall AVF of the entire processor. Assuming the reasonable assumption that logic density is similar among pipeline structures, the probability of a soft error is directly proportional to the area of the structure. Then, the overall AVF is the weighted sum of each individual structure's AVF weighted by the fraction of the structure's area. We use a floor plan of the Alpha 21364 processor [62], illustrated in Figure 3.4, and take area estimates from the floor plan for the structures shown in Figure 3.5 and compute the overall processor AVF.

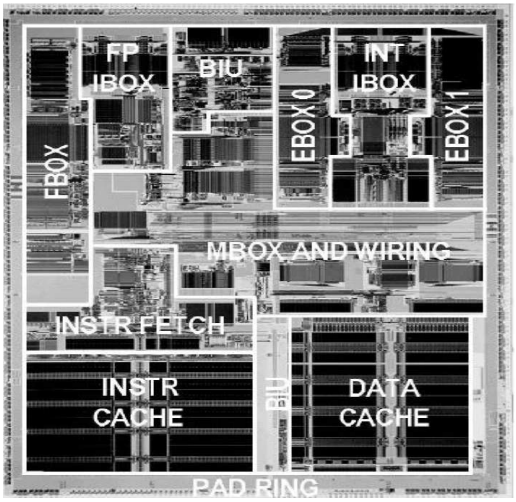
Application	Skipped Instructions	Type
ammp	2,600M	FP
bzip2	1,100M	INT
crafty	500M	INT
equake	400M	FP
fma3d	1,900M	FP
gcc	2,100M	INT
gzip	200M	INT
lucas	800M	FP
mesa	500M	FP
parser	1,000M	INT
swim	400M	FP
vortex	100M	INT

Table 3.2: SPEC CPU2000 Benchmarks.

(a)



(b)



(c)

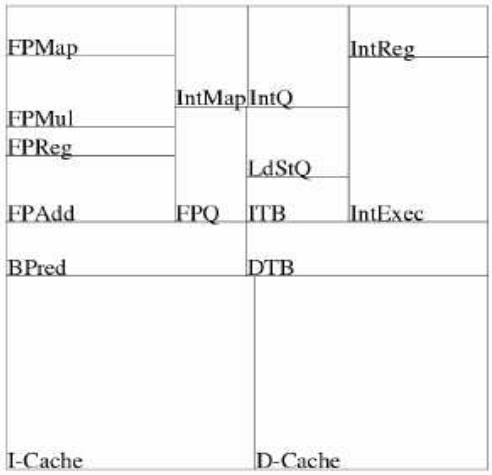


Figure 3.4: Alpha 21364 Floor Plan [62]: (a) Die photograph, (b) Core photograph, (c) Floor plan of the core.

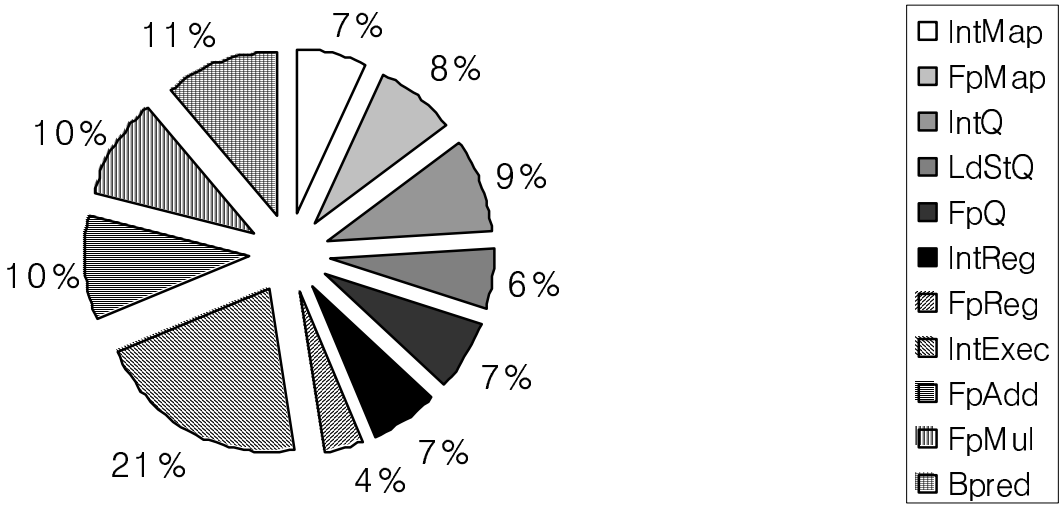


Figure 3.5: Area Model.

Chapter 4

Result Usage Based Filter Checker

This chapter discusses how to implement the proposed AVM technique to a dynamic verification architecture. Instructions that are non-critical from a correctness point of view are identified by a filter checker, described in Section 4.1. Several kinds of congestion avoidance policies are proposed in Section 4.2. The implemented filter checker is evaluated using the methodology described in Section 3.4. Finally, Section 4.3 shows that the proposed AVM has the potential to solve the verification congestion problem.

4.1 Identifying Correctness-Noncritical (CNC) Instructions

The filter checker is implemented to dynamically correlate the correctness-noncriticality of instructions with the congestion avoidance policy described in Section 4.2. The filter checker serves as a first-level checker that filters the verification activity before proceeding to the second-level checker processor. Therefore, the fault coverage loss of AVM is dependent on the speculation for CNC instructions by the filter checker.

We use a CNI (Correctness-Noncriticality Indicator) bit for each instruction to let the second-level checker know about the correctness-noncriticality of that instruction. Since the CNI bit is used only by the checker processor, it is sufficient

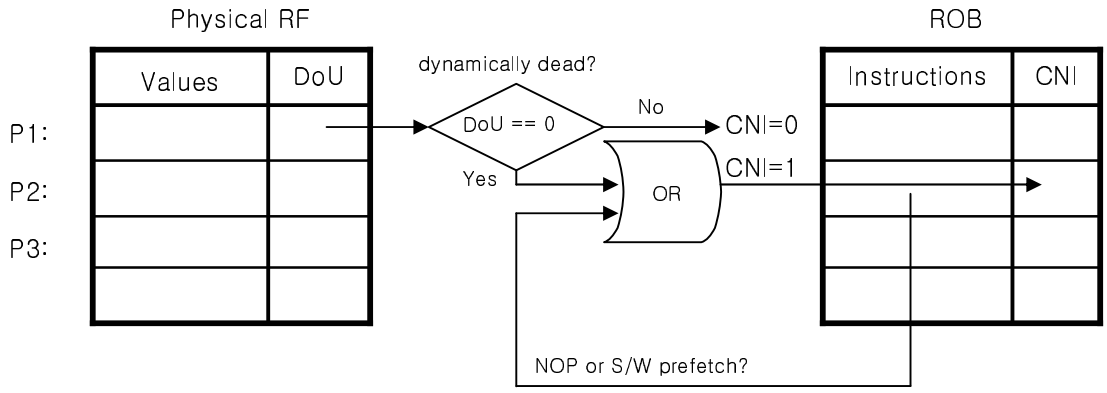


Figure 4.1: Filter Checker Implementation.

that the filter checker can guess and mark CNC instructions during their stay in the reorder buffer of the main processor, just before retiring instructions to the checker processor.

The correctness-noncriticality of an instruction is defined by the fraction of correctness-noncritical bits in the instruction. Correctness-Critical (CC) bits are those that are required for architecturally correct execution, and Correctness-Non-Critical (CNC) bits are those that do not affect correct execution. Our correctness metric uses the Architectural Vulnerability Factor (AVF) metric discussed in Chapter 2. While AVF is used to statically compute the vulnerability factor of a hardware structure, our correctness metric is applied to the dynamic instruction stream for characterizing CNC instructions.

For example, consider NOP instructions. Clearly, the only CC bits in a NOP instruction are the opcode bits that distinguish it from a non-NOP. The remaining bits are CNC. Another example of CNC instructions are the dynamically dead instructions—instructions whose results are not used. Instead of using a dead in-

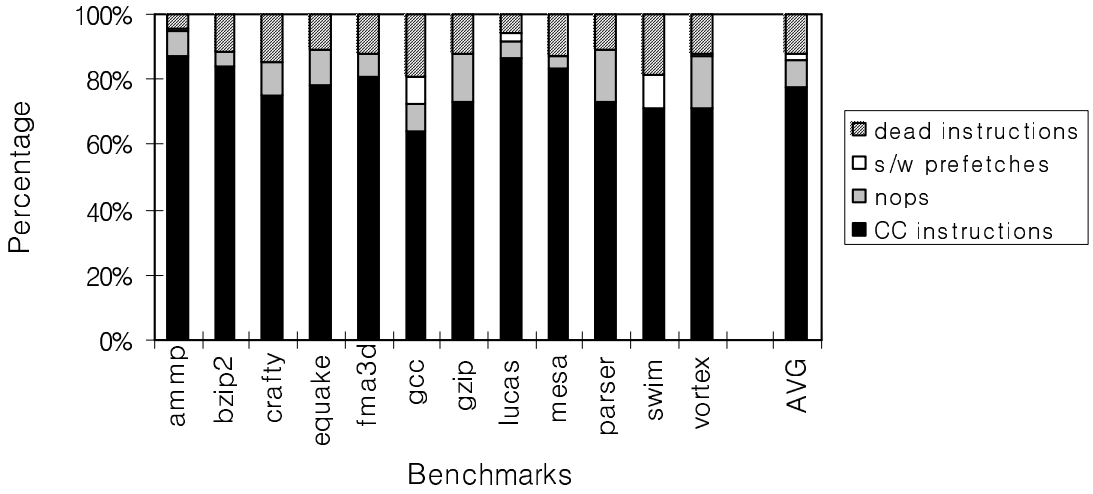


Figure 4.2: Breakdown of Correctness-Non-Critical and Correctness-Critical Instructions.

struction predictor to detect them [15], we use a simple heuristic within the reorder buffer: identify instructions whose results are not read by any other instructions in the reorder buffer and mark their CNI bits to notify the correctness-noncriticality to the second-level checker.

Figure 4.1 illustrates the implemented filter checker. Fortunately, this information is readily available in the reorder buffer because calculating the Degree of Usage (DoU) is straight-forward using the in-order instruction stream at retirement [14]. For each physical register, the filter checker maintains a counter that is incremented when a use of the corresponding register is observed. When a register is overwritten, its counter indicates the degree of usage for the value previously in the register. The counter is then reset and the process resumes. Thus, the degree of usage can be easily exploited to speculate if an instruction is dynamically dead or not.

Our experiment identifying CNC instructions shows 8.8% NOPs for the SPEC2000 benchmarks in Figure 4.2. The filter checker can find up to 12.5% dynamically dead

instructions in our evaluation of the SPEC2000 benchmarks. Therefore, significant performance improvement is possible without losing the fault coverage, if both NOPs and potentially dead instructions occupying 21.3% of the dynamic instruction stream can be marked.

4.2 Congestion Avoidance Policies Simulated

In this section, we categorize congestion avoidance mechanisms and propose both reactive and proactive congestion avoidance policies. When the checker processor becomes a performance bottleneck, AVM can mark some instructions to skip their verification process so as to reduce the congestion at the checker.

- **No AVM (No Marking):** In this policy, no marking is done; the core is just stalled until the congestion is resolved. This is exactly the same architecture as the original DIVA proposal. However, when not using a checker processor with wider bandwidth, this policy showed a severe performance degradation in Section 4.3.1.

- **AVM-FC (Reactive Flow Control Based on Measurement):** In this policy, once a congestion is detected, the verification process is skipped for instructions in a forceful manner. Congestion-phase is tracked by using a performance counter. Based on the measurement of the overall performance, AVM decides whether the congestion happens or not. Once the congestion is over, the system returns to the normal verification mode.

- **AVM-RM (Proactive Random Marking):** To avoid congestion before it occurs, this policy does not need to measure the performance and randomly drops the core's verification service needs. That is why we call it a proactive congestion avoidance scheme. Because a congestion avoidance decision is not correlated with any correctness metric of instructions, the fault coverage decreases.

- **AVM-CM (Proactive Correlated Marking):** This policy uses information on the correctness non-criticality of instructions. The hint is a simple one-bit flag that indicates how likely an instruction is to be correctness-non-critical. The filter checker dynamically identifies Correctness-Non-Critical (CNC) instructions and marks them. Using AVM with this policy, if an instruction is marked as CNC, its verification may be skipped without any loss of fault coverage. Therefore, the loss of fault coverage can be less than the AVM-RM policy discussed earlier.

All of the above AVM congestion avoidance policies trade-off fault coverage for performance in systems or soft computing applications with user-level qualitative interpretation such as multimedia processing where perfect fault coverage is not needed. They can achieve better performance at reasonable design points which provide partial fault coverage. Therefore, with the support of AVM in reliable processors, the fault tolerance design space is enhanced beyond two extreme points, namely perfectly fault-tolerant designs and completely non-fault-tolerant designs.

4.3 Experimental Results

Using the experimental methodology described in Section 3.4, we evaluate the effect of our proposed AVM techniques on the processor performance and the soft error rate.

4.3.1 Sensitivity to Checker's Congestion

Figure 4.3 shows the effect of reduced checker pipeline width. Figure 4.3 (a) shows the normalized Instructions Per Cycle (IPC) relative to 4-way superscalar architecture when the checker's verification widths are 8, 4, 2, and 1. Compared with a 4-way superscalar architecture, the dynamic verification architecture has no performance loss with an 8-issue checker pipeline, but around 10% loss with a 4-issue checker pipeline. As the checker's verification width decreases, the performance degrades severely.

The checker processor pipeline delays the retirement of instructions, forcing the main processor to hold speculative state longer, thus creating back-pressure at retirement. Figure 4.3 (b) shows this back-pressure effect, comparing the fraction of time the reorder buffer (ROB) was full. With decreased verification width, the ROB is full most of the time. If the core's ROB resources fill, the decoder will also stall, as it will not be able to allocate ROB and load/store queue resources. As the congestion continues, the instruction fetch queue gets full in turn. Therefore, these results indicate that the performance of a dynamic verification processor is highly dependent on the bandwidth of the checker's pipeline.

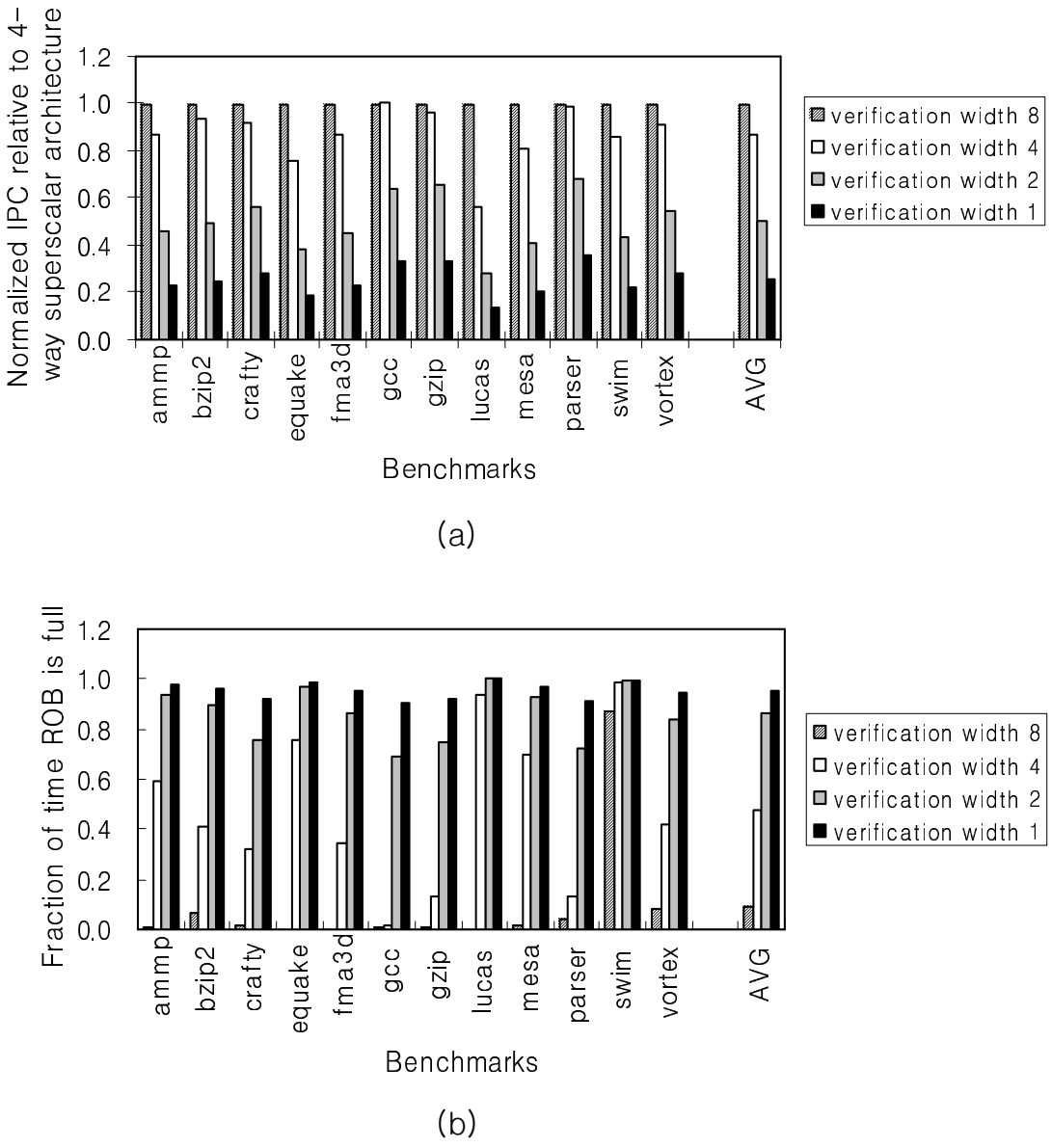


Figure 4.3: Effect of Reduced Verification Width in a Dynamic Verification Architecture: (a) Performance relative to a configuration with no checker processor, (b) Fraction of time the main processor's reorder buffer is full.

4.3.2 Increasing Performance with AVM

One advantage of using AVM is that it increases the performance of fault-tolerant processors. We explore how much the proposed AVM is effective in preventing the checker from becoming a performance bottleneck. Figure 4.4 (a) describes the normalized IPC of several dynamic verification processors (with different marking probability) relative to a non-fault-tolerant processor. It shows how well the proposed AVM manages the verification congestion problem. With no AVM, congestion in the checker badly affects the performance, amounting to 57% of a non-fault-tolerant processor. With increasing marking probability of AVM, the performance of a reliable processor approaches 95% of that of a non-fault-tolerant processor. AVM with 50% marking can increase the performance by 170% compared to a dynamic verification processor without AVM. There are no performance differences between AVM-RM and AVM-CM because the processor performance only depends on the marking probability used in AVM.

Another advantage is that AVM provides a reasonable design point to computer architects. With the support of AVM in fault-tolerant processors, the fault tolerance design space is enhanced beyond two extreme cases—perfectly fault-tolerant and non-fault-tolerant designs. AVM with 100% marking is effectively a non-fault-tolerant superscalar processor. AVM with no marking, on the other extreme, corresponds to full verification. Between these two, a broad range of partial redundancy can be applied according to the amount of verification needs.

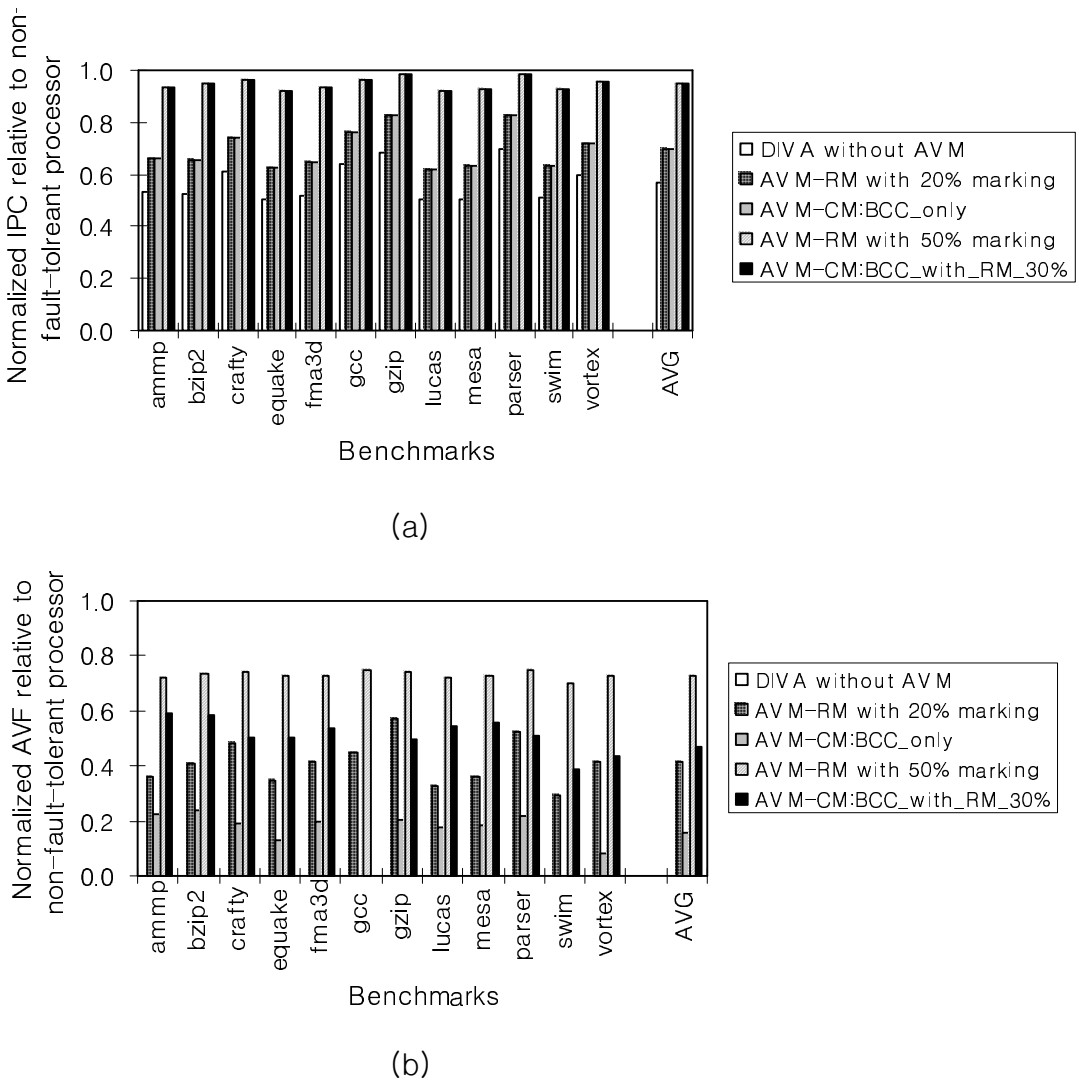


Figure 4.4: Impact of AVM on Performance and Soft Error Rate: (a) Performance, (b) Processor AVF (Architectural Vulnerability Factor).

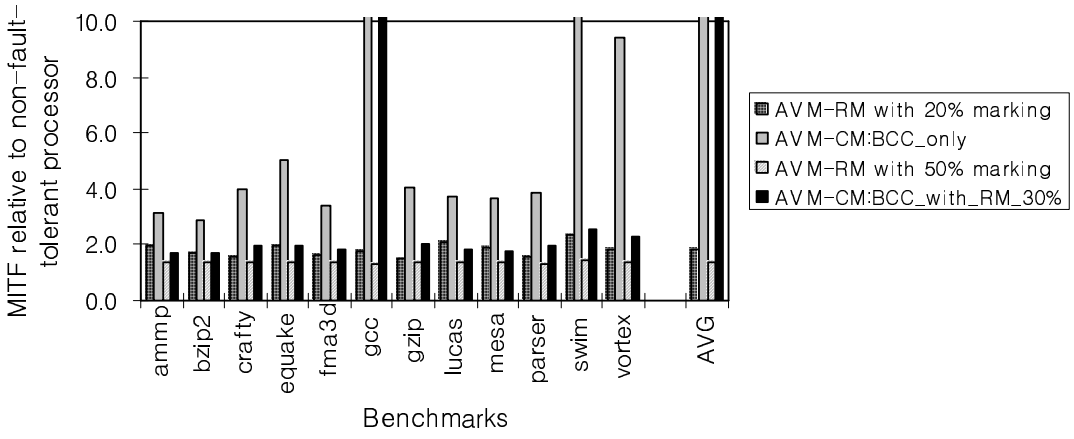


Figure 4.5: Trade-off Between Performance and Reliability.

4.3.3 Better Fault Coverage with AVM-CM

One disadvantage of AVM-RM is that it reduces the fault coverage of a processor and increases the soft error rate. Figure 4.4 (b) shows the normalized AVF of dynamic verification architectures with different marking probabilities and congestion avoidance policies relative to a non-fault-tolerant processor. It shows that the proposed AVM provides better fault tolerance than a non-fault-tolerant processor, but worse than a fully redundant processor. Randomly marking instructions to be skipped naturally decreases the fault coverage. Our proposed AVM-CM congestion avoidance policy (which uses a filter checker), on the other hand, produces only a small decrease in the fault coverage. Figure 4.4 (b) shows that the proposed AVM-CM can reduce the soft error rate by 39% and 67% of AVM-RM for 20% and 50% marking, respectively. Especially, in the case of `gcc`, AVM-CM shows no decrease in fault coverage at all and provides the same level of reliability as a fully redundant processor.

4.3.4 Trade-off Between Performance and Reliability

To validate the utility of the proposed AVM, we need to capture both performance and reliability. For example, a non-fault-tolerant processor has the greatest IPC as well as the greatest AVF, while a perfectly-fault-tolerant processor has the least IPC as well as the least AVF. We can use the MITF metric, discussed in Chapter 2, to reason about the trade-off between performance and reliability. At a fixed frequency and raw error rate of a processor, MITF is proportional to the ratio of IPC to AVF.

A higher MITF means a larger amount of work done by the processor between errors. However, in the case that AVF is equal to zero, MITF goes to infinity and does not capture the performance improvement. Assuming that increasing MITF is desirable within certain bounds, MITF can be used to evaluate the trade-off. Figure 4.5 describes the normalized MITF of the proposed AVM schemes relative to a non-fault-tolerant processor. AVM with correlated 20% marking provides better MITF than the other AVM schemes. The observation that CNC instructions occupied 21.3% of the dynamic instruction stream answers why AVM-CM around 20% marking shows the greatest MITF.

4.3.5 Towards a Zero Performance Penalty Design

By monitoring the committed IPC, AVM can be gated off according to the processor's congestion status. Exploiting the fact that most of the modern processors have a performance counter to monitor the committed IPC, this measurement can

```
1.  #define EPOCH_SIZE              : epoch size
2.  #define eval_perf(n) : evaluate the performance during the nth epoch
3.  For every Tick {
4.      init();
5.      For every EPOCH_SIZE cycles {
6.          perf[epoch_id] = eval_perf(epoch_id);
7.          epoch_id++;
8.          reset_epoch_counters();
9.      }
10.     commit();
11.     writeback();
12.     lsq_refresh();
13.     issue();
14.     dispatch();
15.     fetch();
16. }
```

Figure 4.6: AVM Flow Control Algorithm Pseudo Code.

be fed back to decide if congestion happens or not. For example, if the checker can handle its verification workload, then congestion avoidance mechanisms need not to be applied and AVM can provide perfect fault coverage for the processor. With the feed-backed committed IPCs, our flow control algorithm applies AVM only during the congestion phases of the workload. Figure 4.6 illustrates the pueudo code of the AVM flow control algorithm. Like program phase analysis techniques, our flow control algorithm breaks program execution into a linear sequence of epochs or fixed-size time intervals. For each epoch, the algorithm specifies its congestion status to facilitate the proposed AVM.

The epoch size, measured in processor cycles, is an important parameter for any phase-based technique. As Figure 4.7 (a) shows, the performance is dependent on the epoch size. Based on these experiments, we found that a 10-cycle epoch size yields the same performance as that of a non-fault-tolerant processor. Figure 4.7 (b) presents how the proposed AVM flow control algorithm is effective in that the performance of reliable processors is maintained at the same level as that of non-fault-tolerant processors.

4.4 Summary

The performance of a dynamic verification architecture is very sensitive to the congestion in the checker processor. If the checker processor has limited bandwidth, it cannot always keep up with the main processor. We proposed Active (reactive/proactive) Verification Management (AVM) in an existing CMP imple-

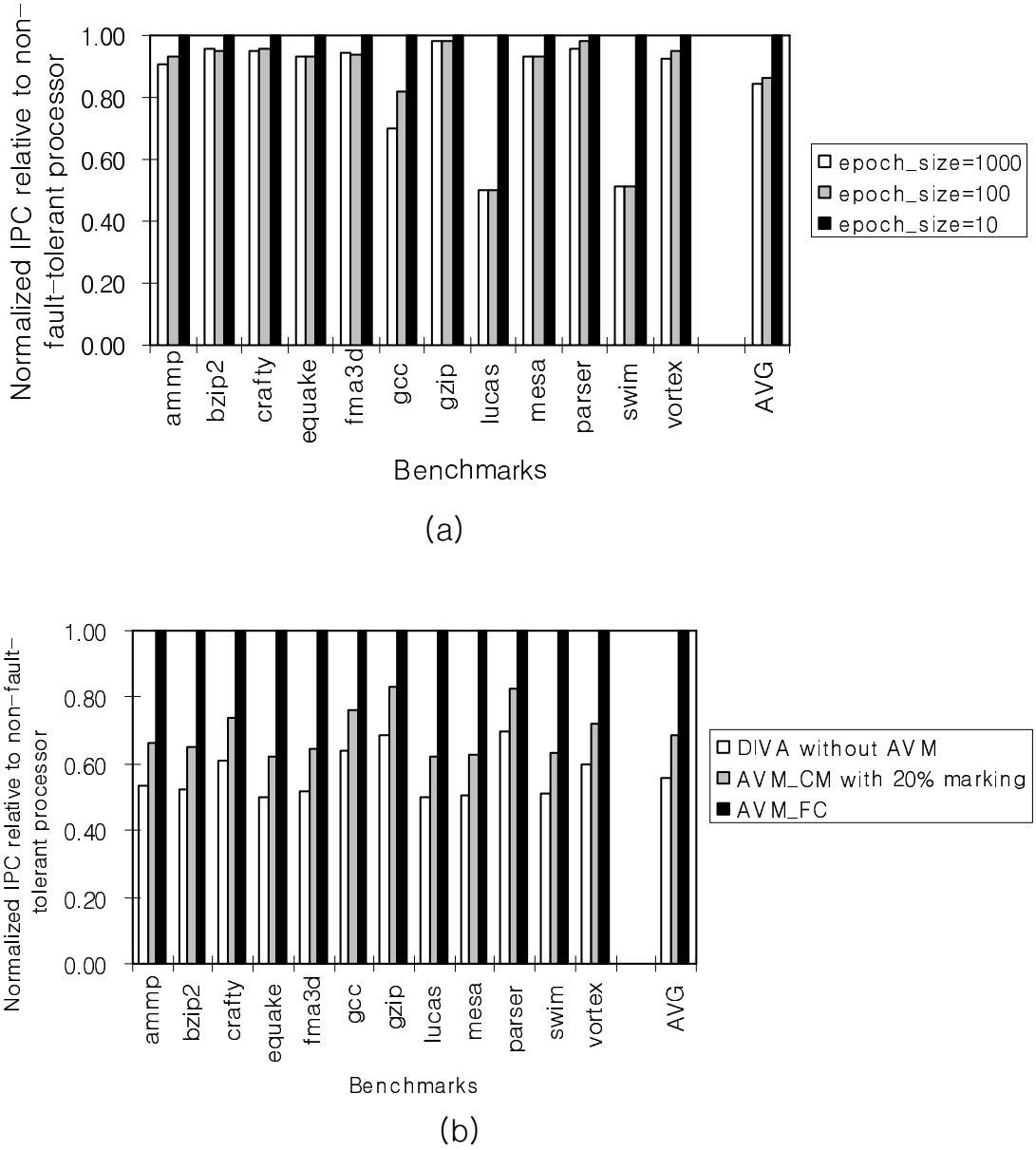


Figure 4.7: Performance of AVM Flow Control Algorithm: (a) Sensitivity of Performance to Epoch Size, (b) Epoch-based Flow Control Algorithm.

mentation to minimize the congestion in the checker, without significantly affecting the fault coverage. We showed promising experimental results when AVM is applied to a dynamic verification architecture. Our proposed AVM is quite effective in preventing the checker from becoming a performance bottleneck, and our correlated marking policy works well in reducing the fault coverage loss.

There is still room for identifying more CNC instructions, including logically masked instructions and operations with narrow-width operands. Once we identify more CNC instructions, the proposed AVM can achieve more performance gain with less reliability loss for better MITF. The proposed AVM can also be exploited to reduce hardware cost and power consumption by halving pipeline width or clock frequency of the checker processor while maintaining the overall performance. To show the effectiveness of AVM in low-power processor design, we plan further study on the impact of AVM on power consumption. With the support of AVM, system designers can choose a reasonable design point concerning the three design metrics of performance, power, and reliability.

Chapter 5

Result Bitwidth Based Filter Checker

5.1 Overview

The previous chapter addressed a performance slowdown problem and presented a binary correctness criticality based filter checker. In this chapter[1], we propose a more effective pro-active verification management approach to mitigate the verification workload for high performance fault-tolerant microprocessors. Before the re-execution process starts at either the redundant thread or the checker processor, the filter checker quantifies a correctness criticality metric, estimating how likely an instruction is correctness-critical and prioritizes the verification candidates. Based on that likelihood, critical instructions may proceed to the second-level checker for further verification, while non-critical instructions may be directly forwarded to the commit stage without further verification.

Computing the likelihood of correctness criticality is accomplished by exploiting information redundancy of compressing computationally useful data bits. Given a 64-bit binary representation, computing '1 + 2' has the same architectural vulnerability to soft errors as executing '334487364720 + 7458523762573', even though the former operation seems to be much simpler. With the support of a value compression technique that explicitly specifies the computationally meaningful bits, the simpler

[1] A condensed version of this chapter is available in [81].

operation could be encoded with a smaller number of bits, effectively reducing its vulnerability to soft errors. In other words, if the information about the number of computationally significant bits in representing the value could be available, the former operation would become less vulnerable to soft errors than the latter. Numerical significance hints let the filter checker prioritize the verification candidates, resulting in less fault coverage loss than skipping them blindly. Figure 5.1 illustrates this idea. Without value compression, all data bytes in Figure 5.1 (a) are numerically significant ones. Figure 5.1 (b) and (c) present two value compression techniques for identifying or replicating numerically significant bits, respectively. Once the value compression is done, the unshaded bytes become numerically insignificant because they are not required for executing a correct computation, while only the shaded bytes are necessary and important. Because the unshaded bytes are computationally meaningless, by compressing numerically significant bytes, soft errors caused by any particle strikes to the unshaded bytes cannot affect the correct operation.

This chapter presents one application of compressing numerically significant bits and of the correctness criticality based filter checker design. This chapter is organized as follows. Section 5.2 investigates whether the number of computationally meaningful bytes is high and presents two value significance compression techniques, which are Value Identification (VI) and Value Replication (VR). Exploiting correctness-criticality metrics for fault tolerance is proposed in Section 5.4. The section defines the Likelihood of Correctness Criticality (LoCC) metric, and presents an LoCC-based filter checker. The proposed LoCC-based filter checker is evaluated using the methodology described in Chapter 3. Finally, Section 5.7 shows

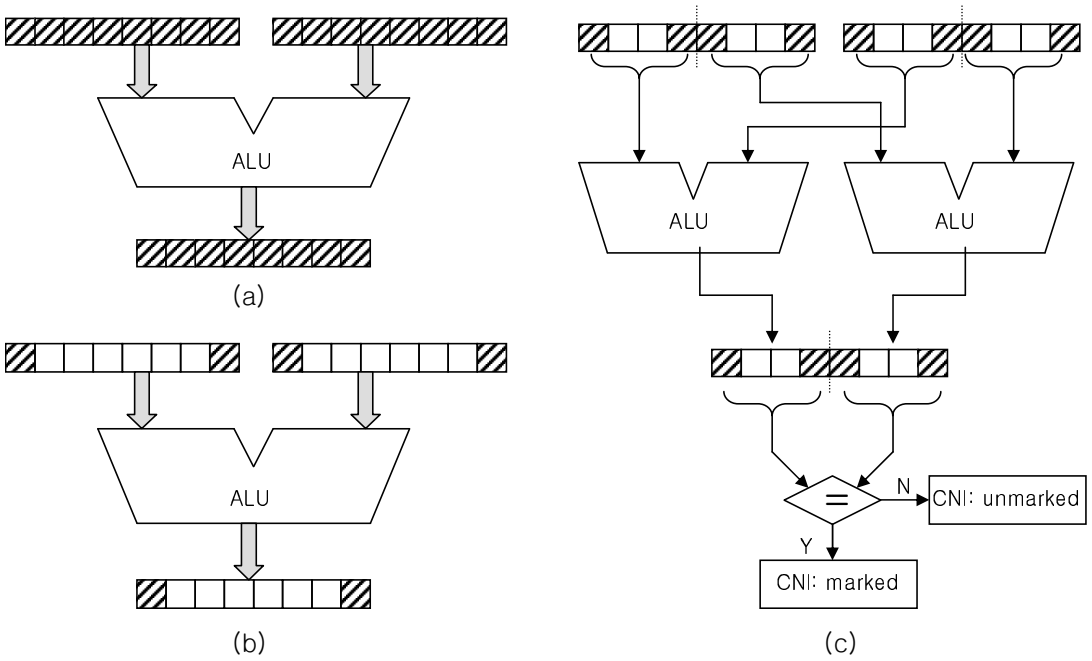


Figure 5.1: Value Compression for Fault Tolerance: (a) No value compression, (b) Value compression for identifying numerically significant bits, (c) Value compression for replicating numerically significant bits.

that the proposed LoCC-based filter checker further improves both performance and reliability, compared to the Binary Correctness Criticality (BCC) based filter checker.

5.2 Significance Compression for Fault Tolerance

Narrow value compression is a special case of compressing numerically significant bits. More precisely, narrow values have a simple high order form—either all zeros or all ones. The information about all zeros or all ones can be encoded with a few bits by a simple compression technique. The distribution of narrow values has been found to be very non-uniform. The observation that a large percentage of the values produced and consumed in a processor are narrow [12, 37, 38], is utilized

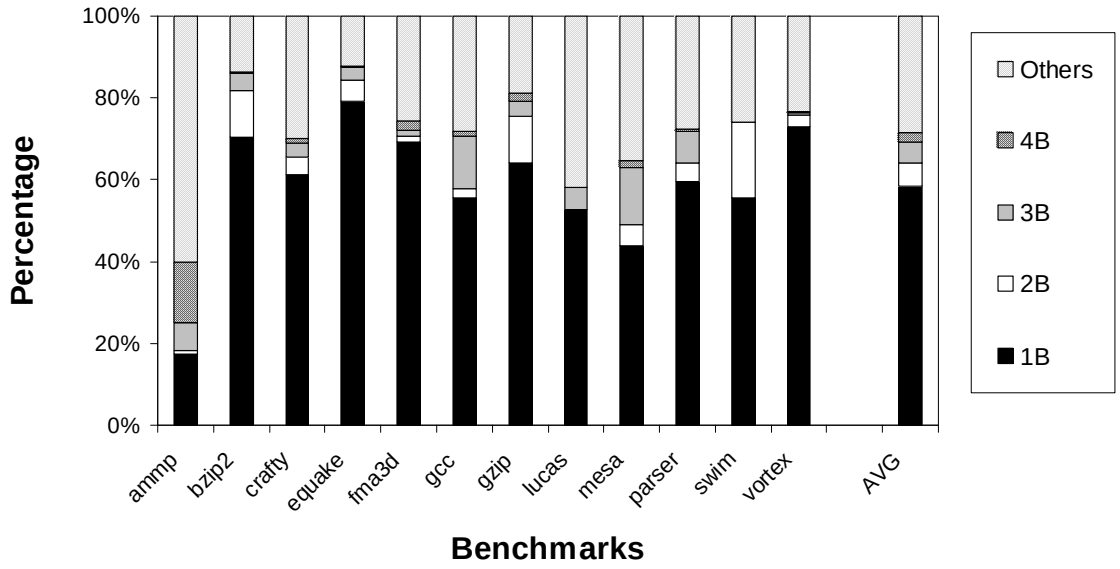


Figure 5.2: Percentage of Computationally Meaningful Bytes.

by several value compression techniques, such as low power pipeline design [17] and cache energy reduction [1, 28, 73]. While previous studies employ value significance compression for low power design, this chapter exploits it for fault tolerance.

To investigate whether the number of computationally meaningful bytes is high, we first measure the distribution of numerically significant bits in the SPEC2000 benchmark suite. Figure 5.2 shows the percentage of computationally meaningful bytes in each program. The data in the figure demonstrate that a wide range of application programs have a large number of narrow values and that the distribution of numerically significant bits is strongly biased. The '1B' label in the legend indicates the percentage of operands that can be represented by one byte. Similarly, '2B' denotes the percentage of operands represented by two bytes. We can see that roughly 60% of all operands are less than 8 bits wide and 36% have just one computationlly meaningful bit! This means that many of the upper order bits in the operand values

are computationally meaningless. Hence, a significance compression technique can reduce the vulnerability to soft errors by storing the information about computationally meaningful bit widths in the conventional value-holding structures such as register file and data cache.

5.3 Exploiting Narrow Values for AVM

We propose a simple mechanism that effectively reduces the vulnerability to soft errors in a processor. It is motivated by the fact that many of the produced and consumed values in a processor are narrow and their upper order bits are meaningless. Therefore, soft errors caused by any particle strike to those higher order bits can be avoided by simply compressing these narrow values. Exploiting these narrow values for increasing both performance and reliability in the AVM model is presented in this chapter. Figure 5.3 shows the proposed AVM architecture with support for value compression capability.

In order to make use of the observed value significance distribution, we propose a significance compression technique for narrow values. To effectively capture instances in which leading zeros or ones are not in multiple of 8 bits, we use a bit granularity instead of a byte granularity. Given a 64-bit architecture, a leading zeros or ones counter logic counts the leading bits and encodes its counting value through a 64-to-6 line priority encoder with inputs exclusive-OR-ed for adjacent bits of the value to determine a bit position where the value changes from 0 to 1 or 1 from 0. The number of leading zeros or ones can be obtained by negating the output of the

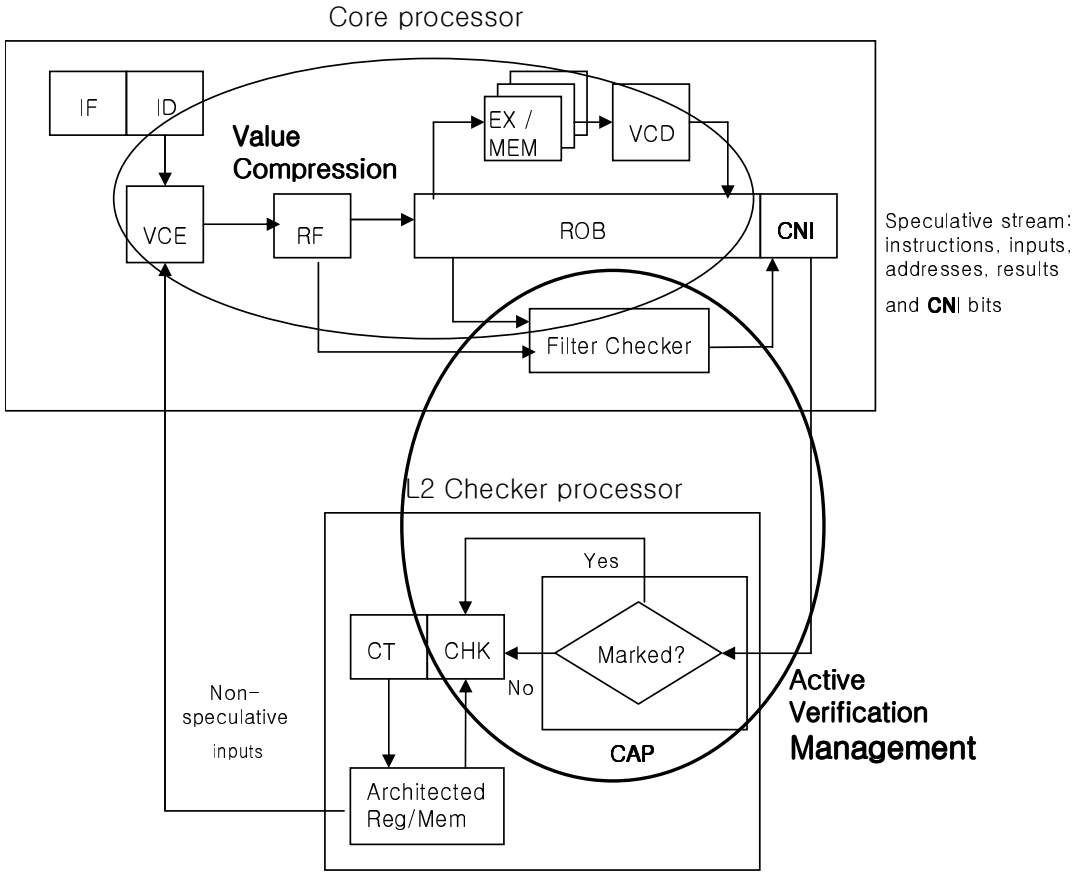


Figure 5.3: AVM Architecture with Value Compression.

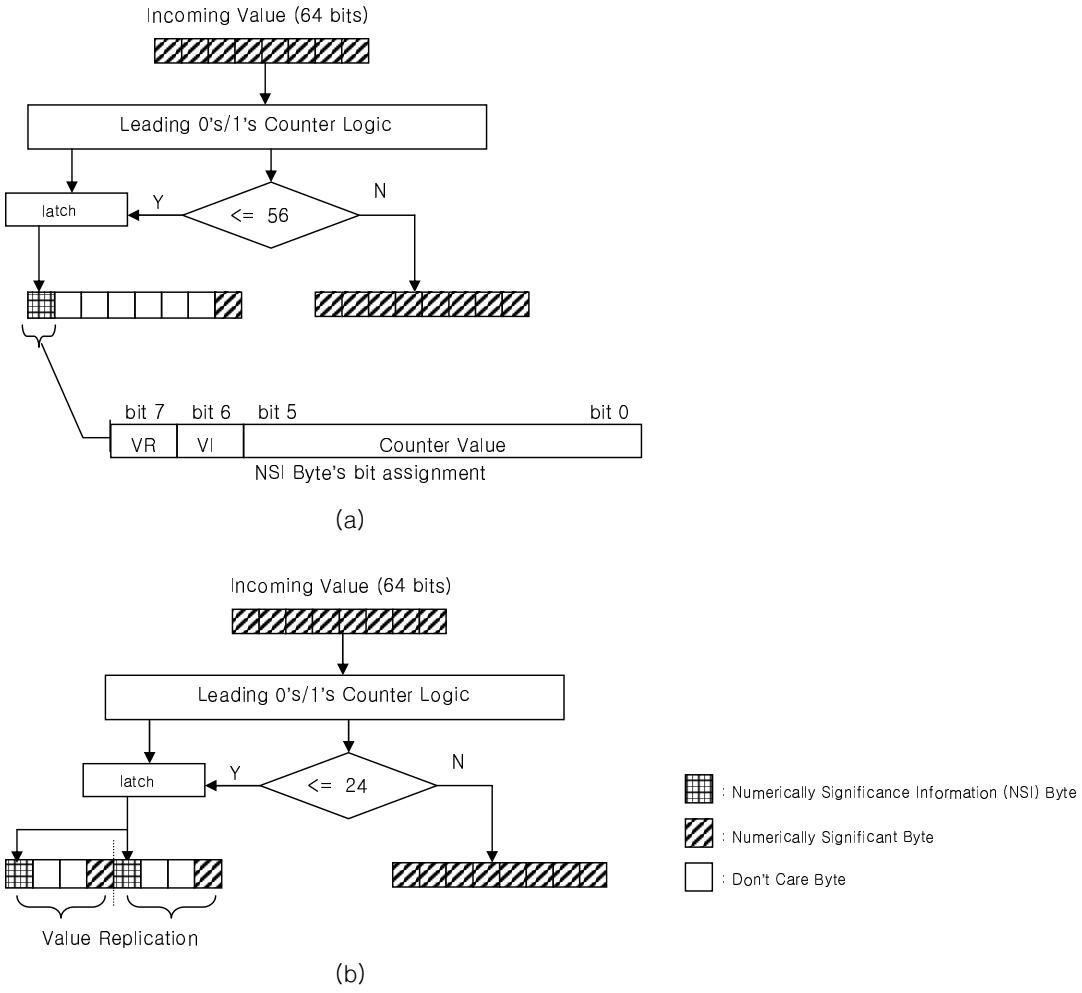


Figure 5.4: Significance Compression Encoder: (a) Value Identification (VI), (b) Value Replication (VR).

encoder. Instead of using additional bits to keep this number, we propose a scheme that reuses the Most Significant Byte (MSB), which can be freed up if the counter value is less than 56.

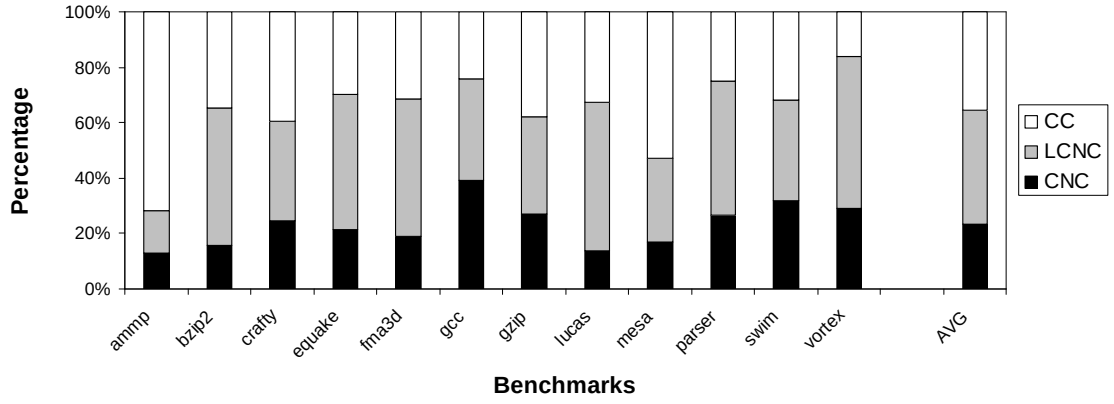
Figure 5.4 illustrates two encoding techniques for narrow values. Figure 5.4 (a) shows a Value Identification (VI) encoding scheme in which a narrow value is defined by a value of width less than 56 bits. The upper two bits of the MSB are narrow value indicator bits to indicate whether the value is encoded or not and which encoding

methods are used for significance compression. The other six bits in the MSB represent the number of leading zeros or ones counted in the previous counter logic. Even though the Value Identification encoding scheme reduces the architectural vulnerability of value-holding structures, it introduces an one-byte overhead. Soft errors can still happen in the unprotected parts of the compressed value. Therefore, we present another Value Replication (VR) encoding scheme to provide redundancy for it by replicating the value (depicted in Figure 5.4 (b)). In this case, a narrow value is defined by a value of width less than 24 bits with the Value Replication encoding scheme.

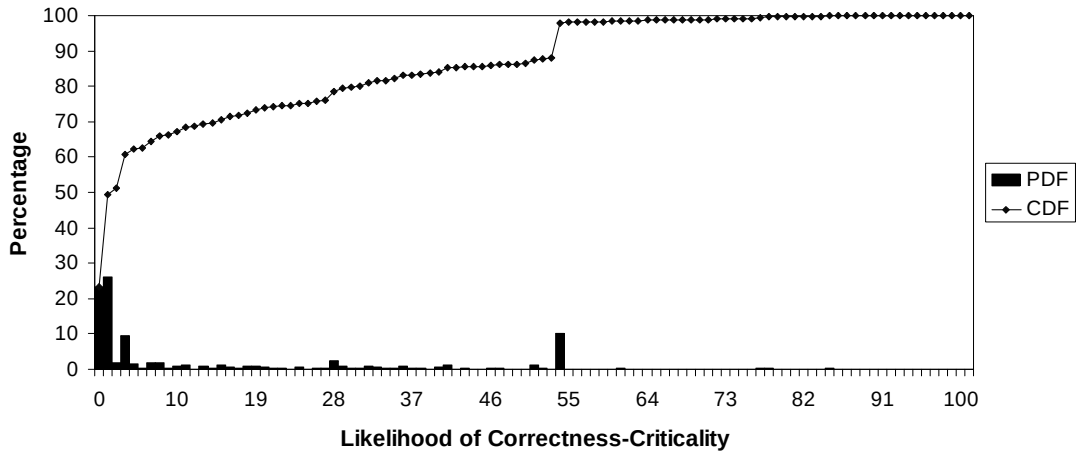
5.4 Computing Value-based Correctness Criticality

This section presents a simple mechanism that effectively reduces the vulnerability to soft errors in a microprocessor, which is motivated by the fact that many of the produced and consumed values in the processors are narrow and their upper order bits are meaningless. Our approach exploits a Value-based Correctness Criticality (VCC) property and manages unnecessary redundancy pro-actively.

With the Binary Correctness Criticality (BCC) metric used in Chapter 4, the number of correctness-non-critical instructions is not very high and only about 20% of the dynamic instructions could be marked. Instead, this chapter uses a Likelihood of Correctness Criticality (LoCC) metric, which indicates how likely an instruction is to be correctness-critical. The filter checker prioritizes the verification candidates based on that likelihood.



(a)



(b)

Figure 5.5: Likelihood of Correctness Criticality: (a) Percentage of LCNC instructions, (b) Density function of LoCC values.

The Likelihood of Correctness Criticality (LoCC) is quantified by computing a Value Vulnerability Factor (VVF), which is defined by the numerically significant bit-width used to compute a result. The VVF metric of a value is defined by the probability that a bit flip caused by a particle strike on any bit in the value will result in an erroneous behavior in the executed computation. Therefore, it can be estimated as a ratio of the number of leading zeros or ones to the total number of bits. The LoCC metric for an instruction is given by a scaled sum of the VVFs of all the source operands of that instruction. With the support of the value compression technique introduced in Section 5.2, the information about the computationally significant bits is already propagated along the data path to compute an execution result. Therefore, we can compute the LoCC metric for each dynamic instruction. The filter checker can utilize this value to speculate how likely the instruction is correctness-critical. Given two instructions I_a and I_b, if $LoCC(I_a) > LoCC(I_b)$, then instruction I_a is more likely to be correctness-critical than I_b. For simplicity, an LoCC of 50% is assigned to an instruction if 50% of the value widths averaged over all source operands of that instruction is computationally meaningful.

5.5 Prioritizing Verification Based on VCC

For the verification priority determined by a filter checker, the computed LoCC values are compared with a threshold LoCC value, which is set to 5 in this chapter, to mark the Likely Correctness Non-Critical (LCNC) instructions. In other words, a type of hypothesis testing with a likelihood function is done to speculate how likely

an instruction is to be correctness-non-critical. Figure 5.5 (b) shows the distribution of the LoCC metric over the SPEC2000 benchmark suite. Its density function shows that the LoCC metric yields a wide distribution of values, which indicates the potential to distinguish not just 0-1 correctness criticality but various degrees of correctness criticality. Figure 5.5 (a) shows the percentage of LCNC instructions determined by a filter checker. Being able to distinguish between CC and LCNC instructions by exploiting the LoCC values (which are equally CC instructions in the case of 0-1 correctness criticality), the filter checker can afford to mark more instructions without sacrificing the fault coverage. To understand the benefit of being able to distinguish between CC and LCNC instructions based on the LoCC values, it is useful to view an LoCC value as a measure of expected penalty of skipping verification when any particle strike happens in the marked instructions. For example, if a 10% fault coverage loss results when a defective instruction with an LoCC value of 50% is skipped for verification, a wrong decision on correctness criticality speculated by a filter checker for an instruction with an LoCC value of 20% would incur a 4% fault coverage loss. By preferring the latter instruction as a marking candidate, the filter checker can save about 6% fault coverage loss.

5.6 Implementation

In order to further reduce the vulnerability to soft errors along the data path, we need a functional unit partitioned into two asymmetric parts which can operate with compressed values. In terms of implementing such a Partitioned-ALU (P-ALU)

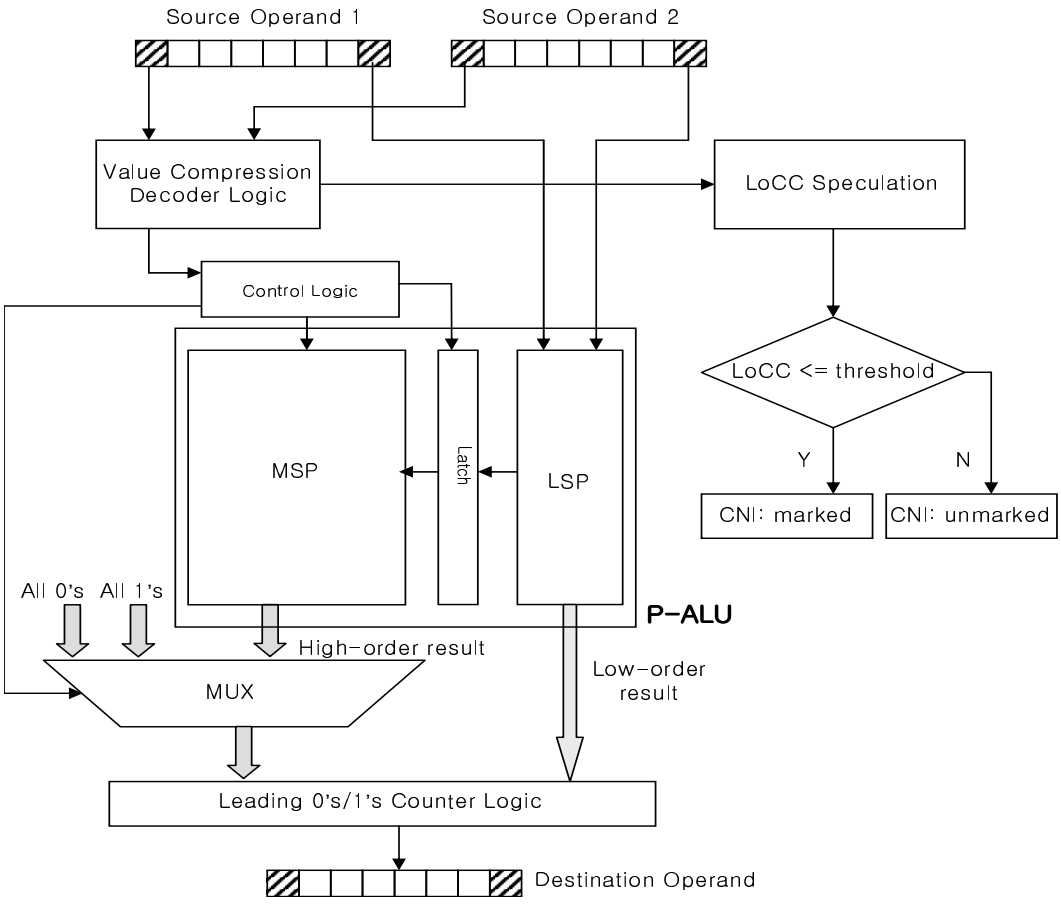


Figure 5.6: A Functional Unit Partitioned into Two Asymmetric Parts.

with value compression capabilities, we borrow from the work done by Choi *et al* [20]. They implemented a functional unit with partially guarded computation. The higher order bits of the input operands are selectively passed on to the MSP part of the function unit that operates on them through latches. The latches are controlled by our leading zeros or ones counter logic, which can detect whether the higher order bits of both the operands are either zeros or ones. By dynamically bypassing the MSP part of the P-ALU to avoid numerically meaningless computation, the vulnerability to soft errors of narrow values can be effectively reduced. Figure 5.6 illustrates how value compression works for the implemented P-ALU.

5.7 Experimental Results

Impacts of our proposed Active Verification Management (AVM) on the processor performance and reliability are evaluated in this section. One advantage of using the proposed scheme is that it reduces the performance degradation due to checker processors. We explore how much the proposed scheme (for AVM with significance compression capability) is effective in preventing the checker from becoming a performance bottleneck.

Figure 5.7 shows the normalized Instructions Per Cycle (IPC) of several dynamic verification processors (with different correctness criticality metric) relative to a non-fault-tolerant processor. For each benchmark, 5 bars are shown, corresponding to the following 5 dynamic verification schemes: DIVA, BCC only, BCC with Random Marking (RM) of skipping blindly 30% of the instructions, LoCC with

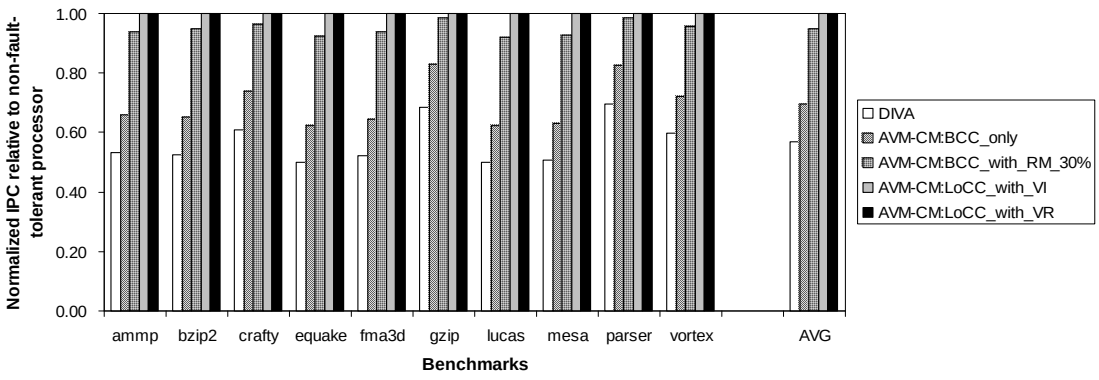


Figure 5.7: Normalized IPC (Instructions Per Cycle) for Five Different Dynamic Verification Schemes.

VI, and LoCC with VR.

Let us analyze the results presented in Figure 5.7. Because DIVA has no marking scheme, its performance is the worst. With no AVM, the checker's congestion badly affects the DIVA processor's performance, reducing it to 57% of that of a non-fault-tolerant processor. The AVM scheme based on 0-1 correctness criticality shows a limited performance improvement over DIVA. The 'BCC-with-RM-30%' in the legend indicates that AVM employs additional random marking of 30% with 0-1 correctness criticality based marking. Through bypassing instructions blindly from verification process, this scheme shows additional performance gain, but the fault coverage decreases compared with the former one. The results for both 'LoCC-with-VI' and 'LoCC-with-VR' schemes demonstrate that the proposed AVM with value compression capability effectively manages the verification congestion problem. VCC-based AVM always maintained the same performance as that of a system without a checker processor, and about 1.7 times the performance of a conventional DIVA processor.

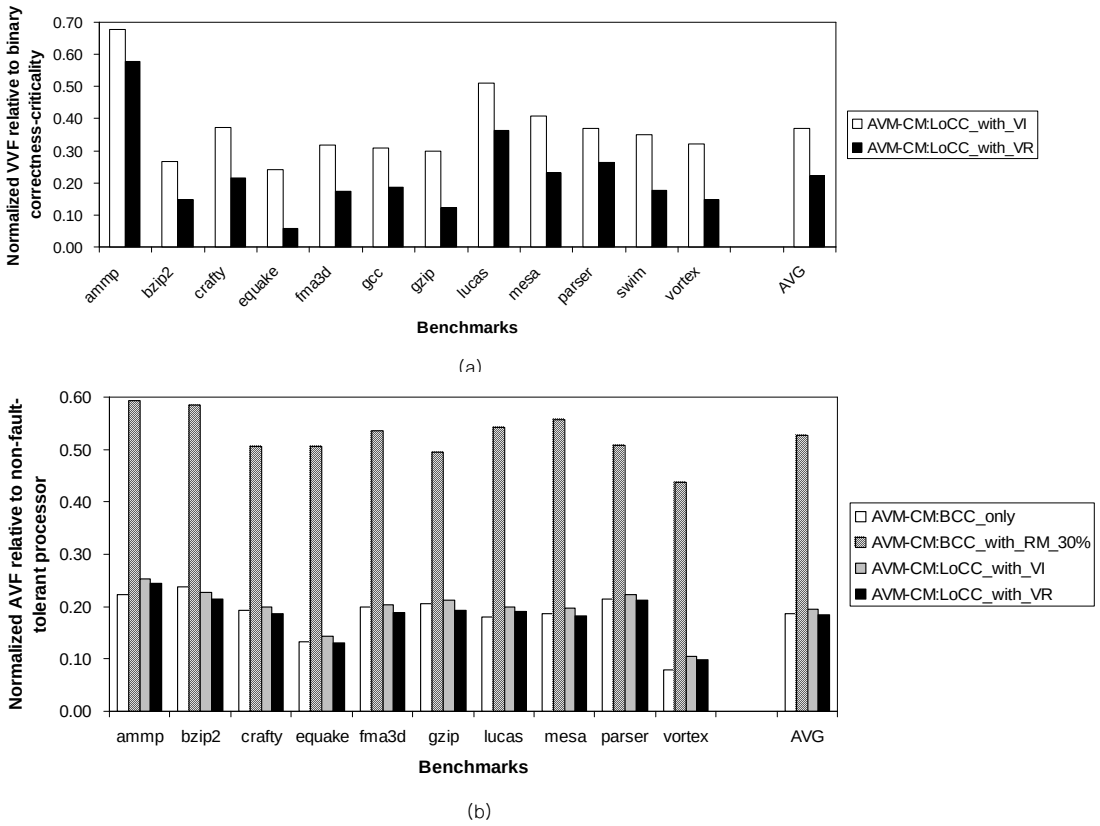


Figure 5.8: Soft Error Rate for Different Dynamic Verification Schemes: (a) Value Vulnerability Factor, (b) Architectural Vulnerability Factor.

According to the definition of the VVF metric, it indicates how much a value is vulnerable to soft errors. After summing each VVF for a value, the average VVF is computed by dividing it by the total number of values. With 0-1 correctness criticality, its VVF is defined by unity because it has no significance compression capability. Figure 5.8 (a) compares the normalized VVF of two significance compression encoding methods relative to 0-1 correctness criticality. Value Replication (VR) shows less VVF than Value Identification (VI). This comes from the fact that the VI scheme introduces an one-byte overhead, which is unprotected from soft errors, while the VR scheme is hardened by replicating the compressed value.

Figure 5.8 (b) shows the normalized processor AVF of AVM architectures with different correctness criticality metrics relative to a non-fault-tolerant processor. It indicates that AVM provides better fault tolerance than a non-fault-tolerant processor. As expected, 'BCC-with-RM-30%' shows severe decrease in fault coverage, compared with the other three AVM schemes, through bypassing instructions blindly from the verification process. It also shows that the proposed AVM scheme based on VCC has little impact on the soft error rate and provides the same level of reliability as the AVM based on 0-1 correctness criticality. Figure 5.8 (b) shows that the proposed VCC-based AVM can reduce the soft error rate by 18% of that of a non-fault-tolerant processor.

To validate the utility of the proposed VCC-based AVM, we need to capture both performance and reliability. We again use the metric *Mean Instructions To Failure (MITF)* to reason about the trade-off between performance and reliability. At a fixed frequency and raw error rate of a processor, MITF is proportional to the ratio of IPC to AVF. A higher MITF means a larger amount of work done by the processor between errors. However, in the case that AVF is equal to zero, MITF goes to infinity and does not capture the performance improvement. Assuming that increasing MITF is desirable within certain bounds, MITF can be used to evaluate the trade-off. Figure 5.9 shows the normalized MITF of several AVM schemes relative to a non-fault-tolerant processor. VCC-based AVM provides better MITF than other AVM schemes based on 0-1 correctness criticality. It shows that the proposed VCC-based AVM can improve both performance and reliability 5.8 times better than that of a non-fault-tolerant processor.

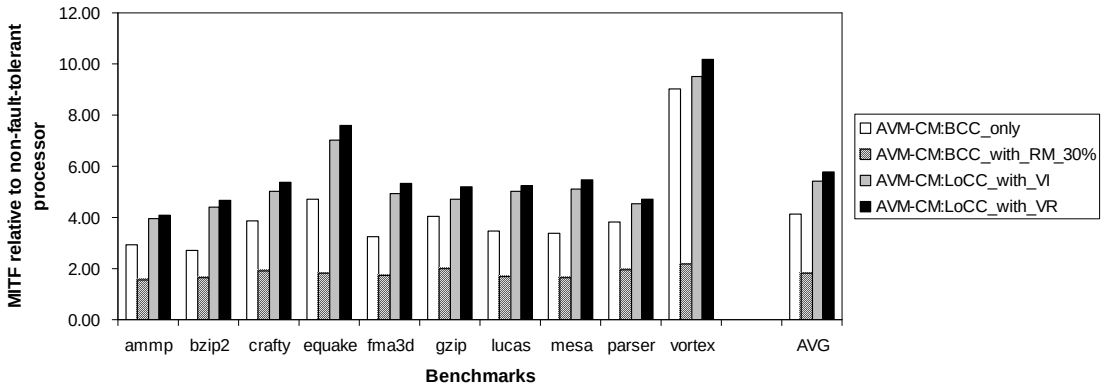


Figure 5.9: Mean-Instruction-To-Failure.

5.8 Summary

This chapter presents a correctness criticality based filter checker, a pro-active verification management technique to mitigate the verification workload of fully-fault-tolerant processors. This is accomplished by exploiting information redundancy of compressing numerically insignificant bits. Exploiting value-based correctness criticality for the filter checker keeps its performance the same as that of high performance non-fault-tolerant processor, while significantly increasing reliability. Experimental results show that the proposed scheme accelerates a traditional fully-fault-tolerant processor by 1.7 times, while it reduces the soft error rate to 18% of that of a non-fault-tolerant processor.

Chapter 6

Result Anomaly Based Filter Checker

6.1 Overview

In the previous chapters, to mitigate the verification workload for fault-tolerant processors, we proposed Active Verification Management (AVM) techniques which employed filter checkers based on correctness-criticality metrics. In this chapter[1], an anomaly-speculation-based filter checker is proposed as a pro-active verification management technique. Anomaly speculation is accomplished by exploiting a value similarity property micro-architecturally, characterized by a frequent occurrence of partially identical values with the same subsets of consecutive bits.

It has been observed that the distribution of used values is very non-uniform. The observation that a few values account for the majority of values used implies that the distribution of data values is strongly biased, which means that instead of assigning verification resources uniformly to every instruction, value locality can be exploited to prioritize verification candidates for an efficient fault tolerance mechanism. The bias in data values is taken advantage of by several micro-architectural techniques, such as value prediction [36], instruction reuse [66], cache hierarchy [32], and efficient register file organization [31, 33], etc. This research especially explores the repeated occurrence of multiple live value instances identical in a subset of their

[1] A condensed version of this chapter appears in [78].

bits, described as a partial value locality [31]. After investigating the partial value locality of instruction streams, we found that the Hamming distance of two consecutive computation results is usually small, indicating that only a few bits in the result value vary among different execution instances. This implies that partial value locality exists in destination operands and its scope is strongly biased.

Based on the biased distribution of Hamming distance, this research investigates further how to exploit the observed partial value locality for soft error tolerance. Because a soft error may incur a transient one-bit flip which cannot be easily detected by monitoring its Hamming distance, this chapter re-defines a micro-architectural similarity distance metric to detect an abnormal deviation away from a nominal similarity distance of computation results, as a special type of partial value locality. This work shows how the value similarity property can be used in the design of a filter checker to lower the pressure on the processor resources, improving its performance with a minimal effect on overall reliability.

Detection of an anomaly in instruction streams is achieved by a speculative value similarity distance cache, followed by an anomaly tester, to mark a re-execution candidate for further verification. The similarity distance cache dynamically captures a run-time behavior of similarity between result values and keeps learning the nominal scope of variance with a confidence score. When an instantaneous variance produced by instruction instances exceeds the nominal similarity distance, a likely-to-be soft error can be marked by the anomaly tester. Figure 6.1 (a) describes a block diagram of the proposed anomaly-speculation-based filter checker. With a value distance predictor and an anomaly tester, the proposed scheme speculates an

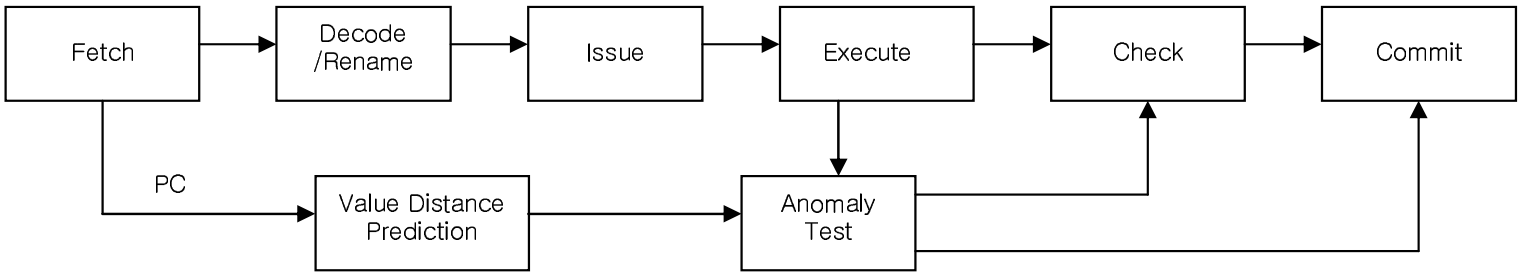


Figure 6.1: Anomaly Speculation Based AVM architecture.

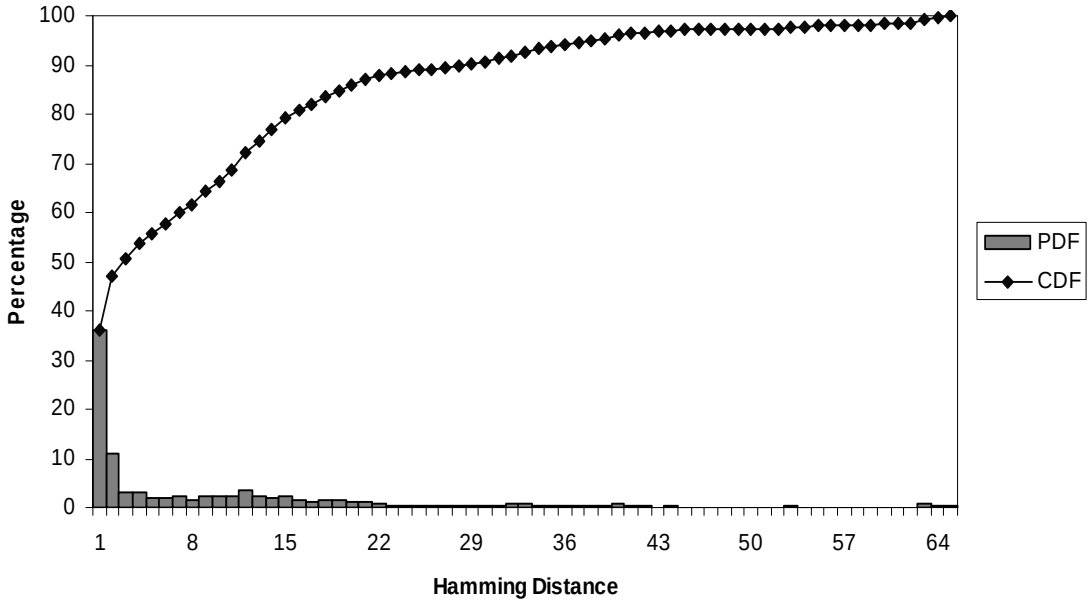
anomaly and provides soft error tolerance based on the anomaly test. This chapter proves the existence and the frequency of occurrence of similar values and presents how the characterized value similarity can be exploited in the design of anomaly-speculation-based filter checker.

This chapter is organized as follows. Section 6.2 investigates a partial value locality and characterizes distributions of both Hamming distance and micro-architectural similarity distance. Section 6.3 describes an anomaly-speculation-based active verification management and implements an anomaly filter checker. Section 6.5 evaluates the impact of the proposed mechanism on performance and reliability. Finally, future work and conclusion are discussed in Section 6.6.

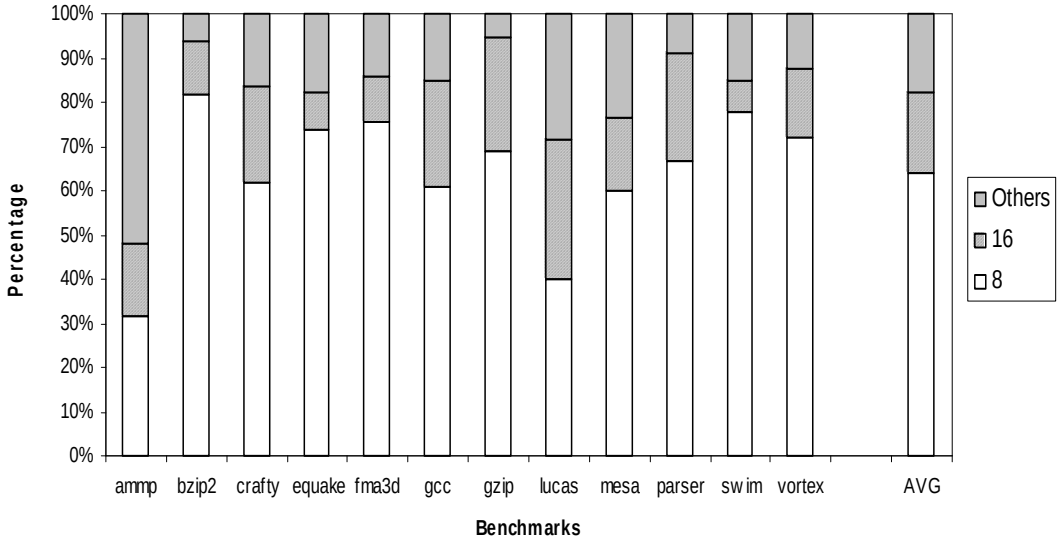
6.2 Characterizing Value Similarity Property

Distance metrics can be used to determine similarity or dissimilarity between two data values. One of well-known similarity distance measures is Hamming distance [23], which is the minimum number of bits that should be modified to convert one value into another. The variance between two data values can be computed by using exclusive-OR on corresponding bits. Its Hamming distance is obtained by the number of 1's in the exclusive-OR bit string. In fact, this is a dissimilarity measure since bigger value shows disagreement in two data values.

Figure 6.2 (a) and (b) shows the distribution of Hamming distance between two consecutive computation result values in the SPEC2000 benchmark suite. The cumulative distribution function indicates a strong bias in the destination operands,



(a)



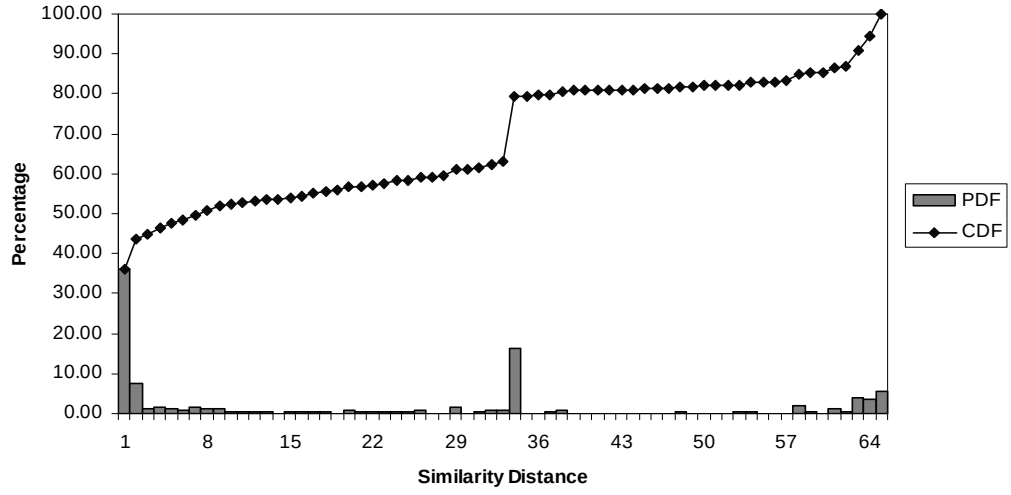
(b)

Figure 6.2: Hamming Distance: (a) Distribution function of Hamming distance, (b) Distribution of n-dissimilar values.

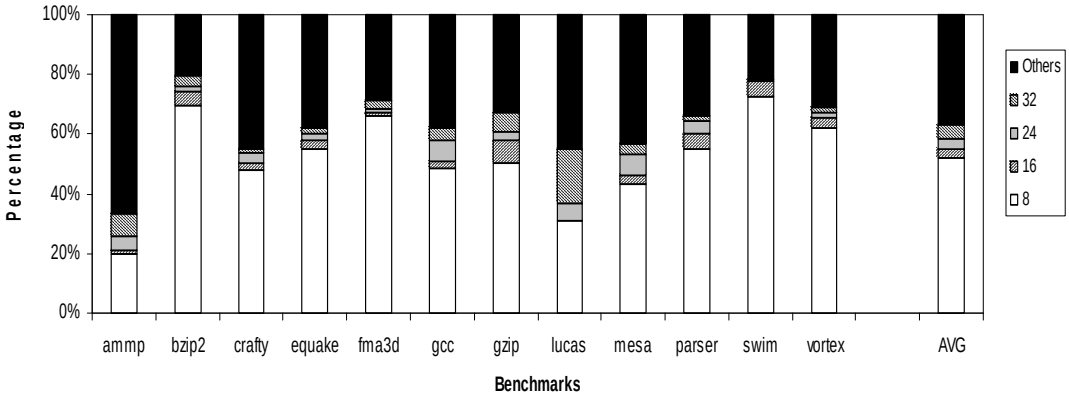
confirming a small range of values account for the majority of values computed. Since Hamming distance is a measure of how much two values disagree each other, the distribution function proves how likely two consecutive results are similar. For instance, it shows that a single value accounts for 36% of all results across SPEC2000 programs. Furthermore, 82% of all values vary only within the scope of 16-bits width, showing that partially identical values occur frequently in a dynamic instruction stream.

Without partial value locality, the distribution would have been uniform. This chapter extends its application to soft error tolerance. Although Hamming distance is a good indicator of how likely values are similar, the metric is not suitable to detect a soft error of one-bit flip in the original data. For example, let's consider a 64-bit binary value 0x0000000000000001 and a bit flip occurs at the most significant bit position. The value then becomes 0x8000000000000001 and then its Hamming distance with previous value 0x0000000000000003 changes from 1 to 2, which is hard to distinguish whether the value is defected. This chapter proposes a new similarity distance measure to easily detect an abnormal deviation from the result values. More specifically, two 64-bit values are called *micro-architecturally n-dissimilar* if they only differ in n least significant bits and are equal in the remaining $(64 - n)$ high-order bits. Micro-architectural similarity distance does not only reflect the fact that two computation results are near one another in the Hamming distance but also captures a deviation scope from one another. Therefore, the proposed similarity distance is a special type of partial value locality.

The result in Figure 6.3 demonstrates that a lot of similar values do exist.



(a)



(b)

Figure 6.3: Distribution of Value Similarity Distance.

Figure 6.3 (b) illustrates the distribution of *micro-architecturally n-dissimilar* values, indicating that the amount of similar values are still high. More precisely, *8-dissimilar* values account for 52% of all result values across the SPEC2000 programs, including 36% of exactly the same values. Different sections of the bar graph show the fraction of *8-dissimilar*, *16-dissimilar*, *24-dissimilar*, *32-dissimilar* and the rest of them, respectively. The cumulative distribution function in Figure 6.3 (a) shows a similar distribution to the narrow value's, presented in the previous literature [12], indicating that there exists a 16% jump from 33 to 34 distance, besides 36% jump from 0 to 1. This is because the memory address in the Alpha ISA uses 33 bits and reflects the fact that memory address operations followed by usual data operations incur a large deviation from the previous result. To simplify the notation, *micro-architecturally n-dissimilar* values will be just called *similar* in the remainder of this chapter.

6.3 Exploiting Similar Values for Anomaly Speculation

Section 6.2 showed the facts that many values are micro-architecturally similar, a group of similar values shares the high-order bits and each value instance in the group is uniquely represented by its remaining least significant bits. Therefore, if the information on the similarity distance would be available before the re-execution process starts, only least-significant bits within the similarity distance could be considered to be computationally useful, while the remaining high-order bits would not be susceptible to soft errors. The characterized value similarity property can be

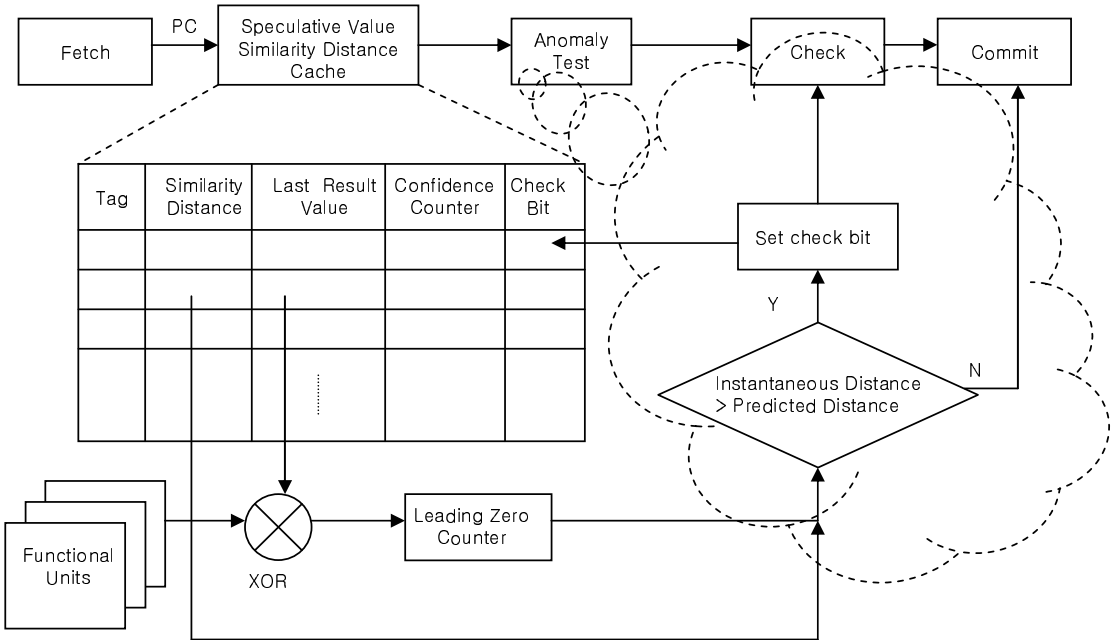


Figure 6.4: Anomaly Test with Speculative Similarity Distance Cache.

exploited to speculate an abnormal behavior due to soft errors during the program execution [79].

Traditional value prediction has a difficulty to predict data values when instructions do not produce strong value patterns like constants or strides. However, although the computed results do not exhibit a predictable value patterns, their similarity distance is likely to be small and stable over time according to our measurement. For example, given instances with result values of 0, 30, 3, 1, 55, in sequence, if the next instance produces an extremely large result such as 100000000000055, it would indicate an abnormal behavior due to a sudden increase of its similarity distance. Therefore, such an abrupt deviation would hint a high possibility of a soft error. Similarity distance also captures a spatial locality, which refers to the fact that memory operations tend to access data in a fixed memory region and generate

a certain address sequence within the same heap space. Therefore, an out-of-range address would indicate a likely-to-be soft error. Even for instructions with a repeating stride pattern, the similarity distance is constrained to the maximum value's width and any result showing a larger distance would signal a potential soft error.

Table 6.1 shows misprediction rate of the proposed anomaly speculation, which is measured by the rate of speculating potential errors when there happens no soft errors. Average misprediction rate is computed by 10.6% across the SPEC2000 programs. Therefore, there seems to be an apparent similarity distance pattern, whose variance is similar nine times in a row, then dissimilar once on average. Anomaly speculation is rather similar to branch prediction than value prediction, in that it predicts whether the similarity distance is normal or abnormal based on the previous similarity history. In general, for branches used to form loops, where a branch is taken many times in a row and then not taken once, a simple 1-bit branch predictor mispredicts at twice the rate that the branch is taken. Similarly, in the proposed anomaly prediction, it seems that we should expect that the accuracy of the anomaly predictor would at least match the frequency of similar values.

6.4 Anomaly Speculation for Active Verification Management

Having demonstrated that the similarity distance has a strongly-biased distribution and are easily predictable, we now present one application of value similarity property to design a pro-active filter checker based on anomaly speculation. The basic concepts of Active Verification Management (AVM) and anomaly speculation

Benchmark	Misprediction Rate	Benchmark	Misprediction Rate
ammp	6.78%	gzip	15.78%
bzip2	10.49%	lucas	3.87%
crafty	15.84%	mesa	8.59%
equake	8.18%	parser	9.10%
fma3d	9.15%	swim	8.99%
gcc	21.72%	vortex	8.72%

Table 6.1: Misprediction rate of anomaly speculation.

are described in this section. The goal of AVM is to reduce the verification workload and to achieve better fault coverage than non-fault-tolerant processor's, without incurring performance degradation even when resource contention happens. The basic idea is similar to that of a cache hierarchy. Because a small cache is fast and a fast cache is expensive, a cost-effective hierarchical solution can be achieved to obtain both fast speed and large size by exploiting the principle of locality. In the AVM, a simple filter checker gives a hint for each instruction whether its re-execution should be done or not. For example, instructions un-marked by the filter checker may be directly passed to the commit stage by skipping verification, while marked instructions may proceed to the second-level primary checker for further re-execution.

This chapter presents an anomaly-speculation-based filter checker in the AVM design paradigm. Whenever an instruction produces a result that deviates a lot from the previous result, it would hint a potential soft error due to such an abnormal behavior. With the hint given by an anomaly test, only the marked instructions

proceeds to further verification, otherwise the computed result can be committed to the physical register file. Given a similarity distance n, any error in the least significant n bits cannot be detected by the proposed anomaly speculation, compared to fully-fault-tolerant architectures. However, the majority of data paths producing $(64 - n)$ high-order bits of the computed results can be protected, which can significantly reduce Architectural Vulnerability Factor (AVF) of the processor.

The proposed design for anomaly speculation is illustrated in Figure 6.4. There are two major structures in the anomaly-speculation-based filter checker, which are a speculative similarity distance cache and an anomaly tester. The speculative distance cache dynamically tracks value similarity of instruction streams and keeps learning its run-time behavior with a saturating counter of confidence score. The similarity distance cache can be accessed by program counter of each instruction. A current execution result is exclusive-OR-ed with the previous result to compute an instantaneous variance. Its similarity distance is obtained by a Leading Zero Counter (LZC) to count the number of consecutive zeros in the high-order bits. Then, an anomaly tester compares the current similarity distance with the speculated one in the similarity distance cache. If the current similarity distance exceeds than the speculated one, the confidence score should be considered next to check whether it is still learning a nominal scope of value similarity. If the confidence has the maximum score, a potential soft error is detected and a check bit flag is set, otherwise the speculated similarity distance is replaced by the current one and the confidence score is reset by zero. If the current similarity distance is smaller than the speculated one, the confidence score is incremented and no soft error is speculated. The previous

result field is then updated by the current one.

6.5 Experimental Evaluation

For the core processor, a 4-way 64-bit superscalar processor is used. The primary checker processor has a 4-issue checker pipeline, which is an in-order single CPI machine with its own register file, functional units, and L1 cache of the same type as the core processor. The speculative similarity distance cache has 256 entries and is configured as 2-way set-associative. Each entry is composed of several fields, which are a 6-bit speculated similarity distance, a 64-bit last result value, a 2-bit confidence score and a 1-bit check flag.

Figure 6.5 (a) shows dramatic performance slowdown of fully-fault-tolerant processor due to the resource contention, compared with non-fault-tolerant superscalar processor. One advantage of using the proposed anomaly-speculation-based filter checker is that it removes the performance degradation due to the resource contention from redundant re-execution, increasing performance relative to fully-fault-tolerant processors. The performance degradation result was already introduced in Chapter 4. The question is that how well the proposed verification management works. Figure 6.5 (b) shows the Instructions Per Cycle (IPC) relative to fully-fault-tolerant DIVA-like architectures. The IPC of proposed anomaly-speculation-based AVM is 1.8 times larger on average across the SPEC2000 programs, than that of fully-fault-tolerant processor. Furthermore, the proposed scheme has the same IPC as that of non-fault-tolerant 4-way superscalar processor. Through bypassing in-

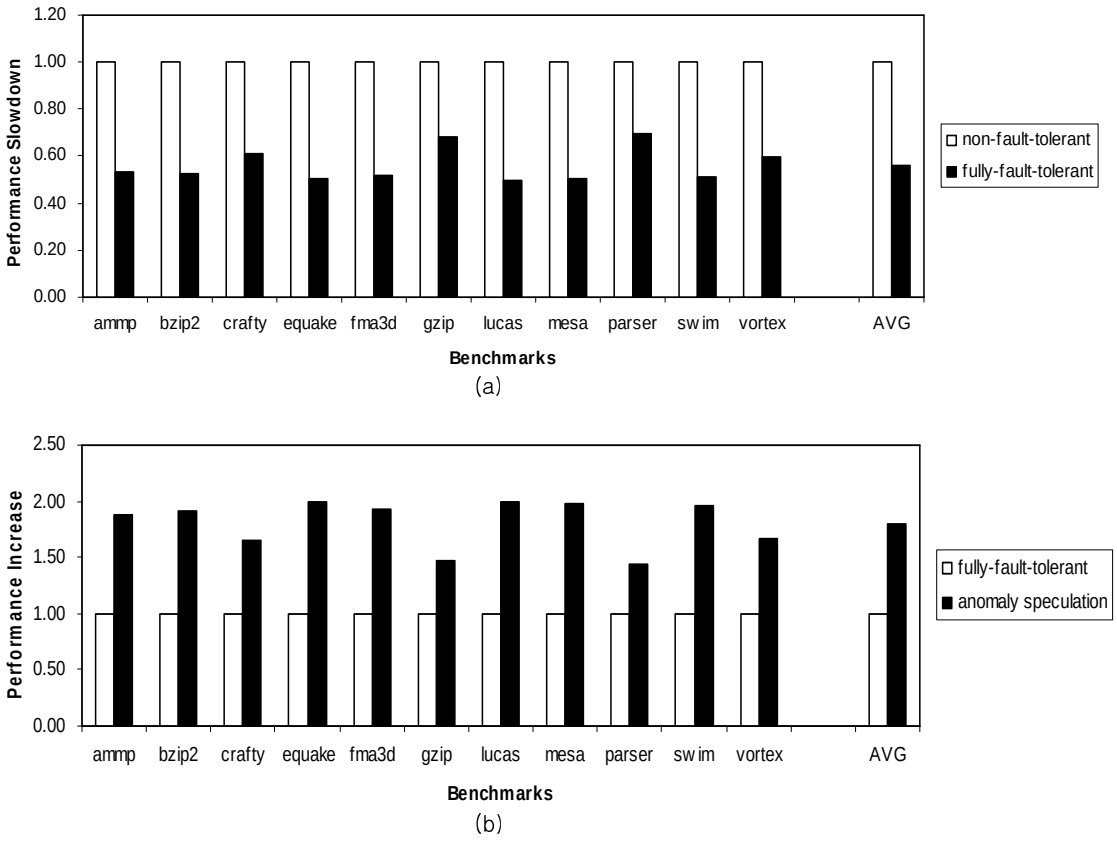


Figure 6.5: Relative IPC.

structions without abnormal results from further re-execution process, the result in Figure 6.5 demonstrates that our anomaly-speculation-based scheme works well to completely remove the performance degradation happened at fully-fault-tolerant processors.

Another advantage is that the proposed anomaly-speculation-based filter checker decreases Soft Error Rate (SER) relative to non-fault-tolerant processors. To evaluate the impact on reliability of the processor, we used the same methodology of computing Architectural Vulnerability Factor (AVF), described in [45] The processor's SER is computed by the product of raw error rate and AVF of the hardware structure. Because raw error rate is generally likely to be proportional to the area of each hardware component, the processor's AVF is obtained by a weighted sum of each component's AVF. Given a floor plan of the Alpha 21364 processor [62], we used the area model to compute the weight.

Any soft error in the least significant bits, within the scope of the similarity distance, cannot be detected by the proposed anomaly-speculation-based filter checker. Compared to fully-fault-tolerant architectures, this refers to decrease fault coverage of the proposed scheme. However, majority of data paths producing high-order bits can be protected from soft errors in its upper bits out of similarity distance. Figure 6.6 shows the processor AVF relative to non-fault-tolerant processors. The AVF of anomaly-speculation-based processor is 53% smaller on average across the SPEC2000 programs, than that of our baseline superscalar processor. Therefore, the result in Figure 6.6 demonstrates that the potential soft error, indicated by the proposed anomaly speculation, can significantly reduce AVF of the processor,

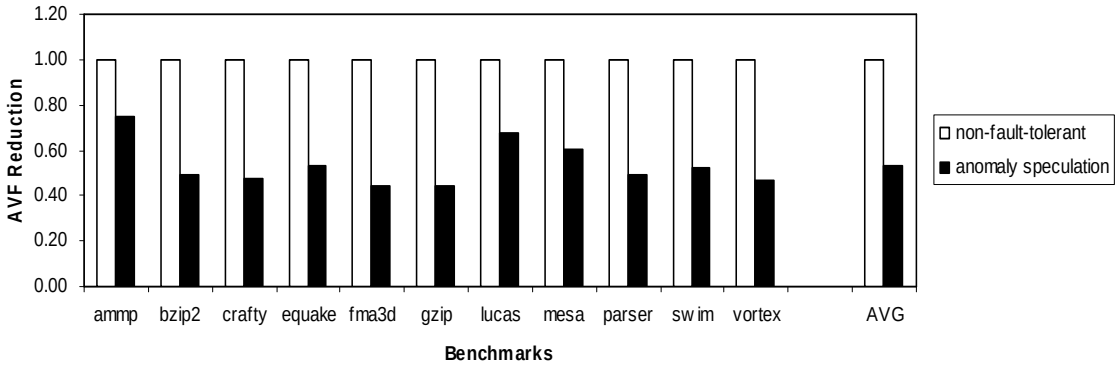


Figure 6.6: Relative Architectural Vulnerability Factor.

resulting in improvement of reliability.

6.6 Summary

This chapter presented an anomaly-speculation-based fault tolerance, a proactive verification management technique to mitigate the verification workload of fully-fault-tolerant processors. This was accomplished by exploiting value similarity property. Experimental characterization proves the existence and scope of similar values. Results demonstrate that the proposed scheme increases 1.8 times faster than that of fully-fault-tolerant processor with a minimal impact on overall soft error rate and reliability of the proposed AVM processor is 53% better than that of non-fault-tolerant processor. As future work, we plan to investigate how to exploit narrow values for soft error tolerance, exploiting the fact that it is a subset of similar values.

Chapter 7

Comparative Analysis and Potential Applications

7.1 Comparison of the Proposed Filter Checkers

Impacts of our filter checkers proposed in this dissertation on the processor performance and reliability are evaluated in this section. One advantage of using the proposed scheme is that it reduces the performance degradation due to checker processors.

Figure 7.1 illustrates the normalized Instructions Per Cycle (IPC) of three filter checkers relative to a non-fault-tolerant processor. For each benchmark, 4 bars are shown, corresponding to the following 4 dynamic verification schemes: No AVM, result-usage-based filter checker, result-bitwidth-based filter checker, and result-anomaly-based filter checker. All three schemes show better performance than a dynamic verification processor without AVM and the proposed AVM are very effective in preventing the checker from becoming a performance bottleneck. Furthermore, either result-bitwidth-based or result-anomaly-based filter checker maintains the same performance as that of non-fault-tolerant processor, and about 1.7 times the performance of a conventional DIVA processor.

Figure 7.2 shows the normalized processor AVF of three schemes relative to a non-fault-tolerant processor. It indicates that AVM provides better fault tolerance than a non-fault-tolerant processor. Among filter checkers, 'result-anomaly-based'

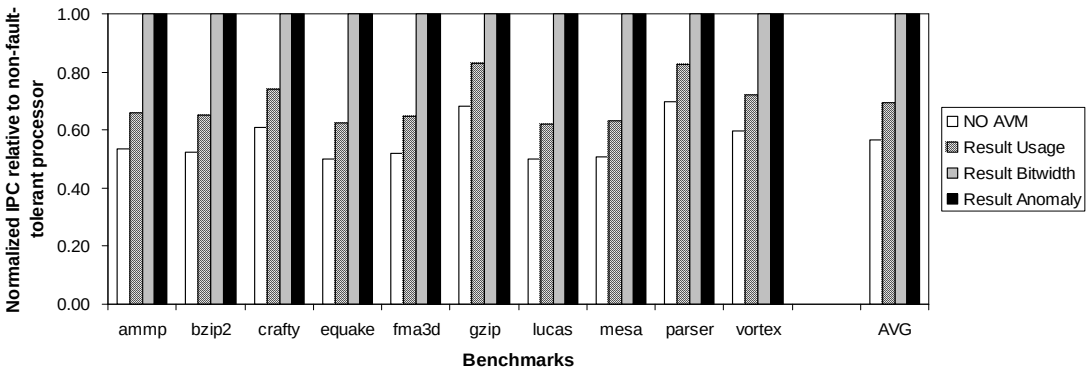


Figure 7.1: Normalized IPC (Instructions Per Cycle) for Three Filter Checkers.

filter checker shows worse fault coverage, compared with the other two filter checkers. However, those based on the correctness-criticality metrics have almost the same level of soft error rate, which reduces the soft error rate by 18% of that of a non-fault-tolerant processor.

Similarly, Figure 7.3 shows the normalized MITF of three filter checkers relative to a non-fault-tolerant processor. The result-bitwidth-based AVM provides better MITF than other AVM schemes. It shows that the proposed result-bitwidth-based AVM can improve both performance and reliability 5.8 times better than that of a non-fault-tolerant processor.

7.2 Reliability and Complexity Impact of the Filter Checker

Several issues related to the implementation of the filter checker are elaborated in this section. We address the hardware complexity, the effects of soft errors in the filter checker itself, and the impact on cycle time. The filter checker needs extra hardware to identify whether instructions are correctness-noncritical and requires

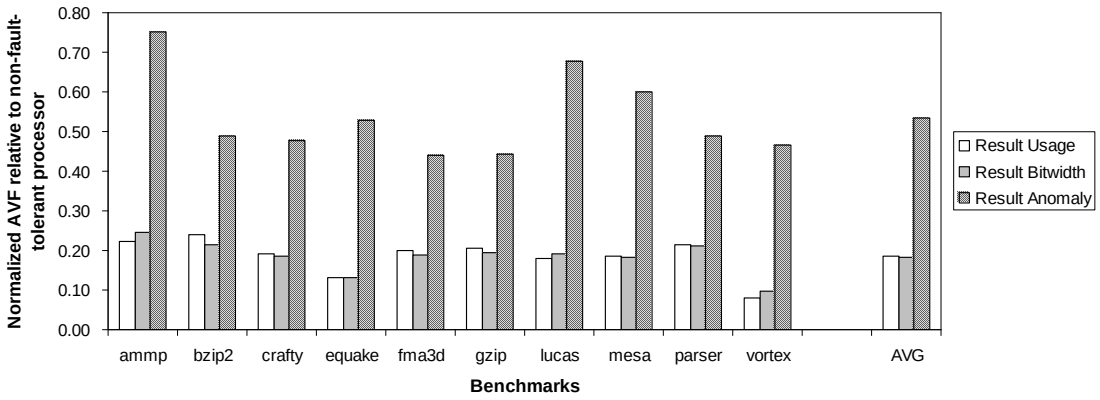


Figure 7.2: Soft Error Rate for Three Filter Checkers.

some extra fields such as Degree of Usage (DoU) in the register file and Correctness Non-criticality Indicator (CNI) flag in the reorder buffer. However, this hardware overhead is minor compared to a simple duplication scheme. The Active Verification Management (AVM) is a reasonably good fault tolerant technique to provide high performance, with only moderate additional hardware in the form of a simple checker processor and a filter checker.

Like any logic units in the processor, the filter checker is also vulnerable to soft errors and it can cause the miss-speculation of correctness-noncritical instructions. However, there is no need for any protection in the filter checker. The reason is that soft errors that corrupt the filter checker have two possible outcomes—they induce the filter checker to signal a false CNI information, resulting in either verification overhead in the second-level checker or a loss of fault coverage.

The filter checker only needs the instruction's opcode and operands to start a CNI decision. This information is available as early as the decode stage, while the only requirement on the filter checker is that the CNI decision is completed by the

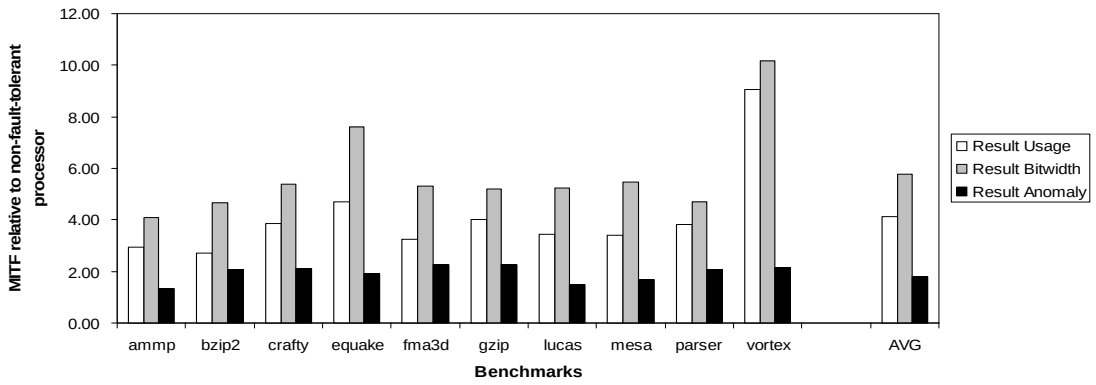


Figure 7.3: Mean-Instruction-To-Failure for Three Filter Checkers.

end of execution stage. Therefore, the filter checker is not on the critical path of the processor and should not badly impact on the cycle time.

7.3 Other Potential Applications

This section presents potential applications for utilizing the proposed AVM. For dynamic verification to be viable, the checker processor bandwidth must match the retirement bandwidth of the core processor. The checker processor can slow the progress of the core processor in the following design examples.

One example is a reliability-preferred design through a more robust checker [4]. In an effort to build an electrically robust implementation of the checker, it may be necessary to construct it with large timing margins and large transistors to resist noise and tolerate natural radiation interference. It will increase the checker's latency, which can delay the retirement of instructions. These delays may cause congestion in the instruction window, effectively reducing the instruction window size and amount of Instruction Level Parallelism that can be exploited.

Another important example is a power-preferred design through low-power checker [49] or aggressively low supply voltage [3]. The need for redundancy in reliable processors is directly opposed to the growing demand [11, 41] for more power efficient operation. Conventional techniques [4, 44, 51, 54, 68, 72] that provide fault detection by supporting whole-thread duplication generally incur significant energy overhead, which can exacerbate the already severe problem of power consumption and heat dissipation. Approaches that supply the necessary level of robustness at a given throughput requirement may also be power-aware. For example, through the use of Dynamic Voltage Scaling (DVS), energy overhead can be reduced significantly. However, it turns out that the maximum operating frequency of the checker processor is limited by the scaled supply voltage [41], which causes severe performance degradation. Furthermore, if it is applied to the power-preferred design through low-power checker described previously, it is possible to reduce the power consumption as well as increase the performance with a small loss in fault coverage.

Therefore, the proposed AVM could be utilized by the following application areas. First, the low power reliable processor design is one promising area to consider. The verification bandwidth can be scaled by gating off unnecessary verification pipelines according to the processor's congestion status. For example, during a low-IPC phase in the main processor, a narrower verification width is sufficient to check the computations of the main core. Such a variable verification bandwidth scaling and adjusting the marking probability with the committed IPC are adaptive approaches to reduce power consumption in reliable processors.

A simple example for low power design is illustrated in Figure 7.4. Here we

can slow down the checker processor by using Dynamic Frequency Scaling (DFS) or Dynamic Voltage Scaling (DVS) and reduce the power overheads of either RMT or DIVA-like microprocessors. DFS is a popular power reduction techniques as dynamic power is directly proportional to operating frequency. Dynamic power P is given by the following equation:

$$P = a \times C \times {V_{dd}}^2 \times f \tag{7.1}$$

where a is a switching activity factor, C is a capacitance being charged, V is a voltage swing, and f is a processor frequency.

In Figure 7.4, (a) shows a $2GHz$ $150W$ DIVA processor showing performance slowdown with $IPC = 2$ due to the checker's congestion. After applying DFS to the original DIVA, in which the checker's frequency gets reduced to $1.33GHz$, the low-power DIVA, illustrated in (b), achieves less power $133W$ than (a), but its performance degrades a lot to $IPC = 1$ as the checker's congestion gets worse than (a). Figure 7.4 (c) illustrates that the proposed AVM is added into the low-power DIVA after applying DFS. With a good AVM marking, (c) can not only increase performance but also reduce power. In our example, it can achieve $IPC = 3$ and $P = 133W$ with a $1.33GHz$ checker processor.

Another promising application is to use the AVM for adaptive resource allocation in simultaneous multi-threading. By exploiting the congestion avoidance concept introduced in this paper, an efficient resource sharing scheme can be designed. For example, suppose that two threads share equal number of entries in the reorder buffer. One thread can have more issue rate than the other. In that

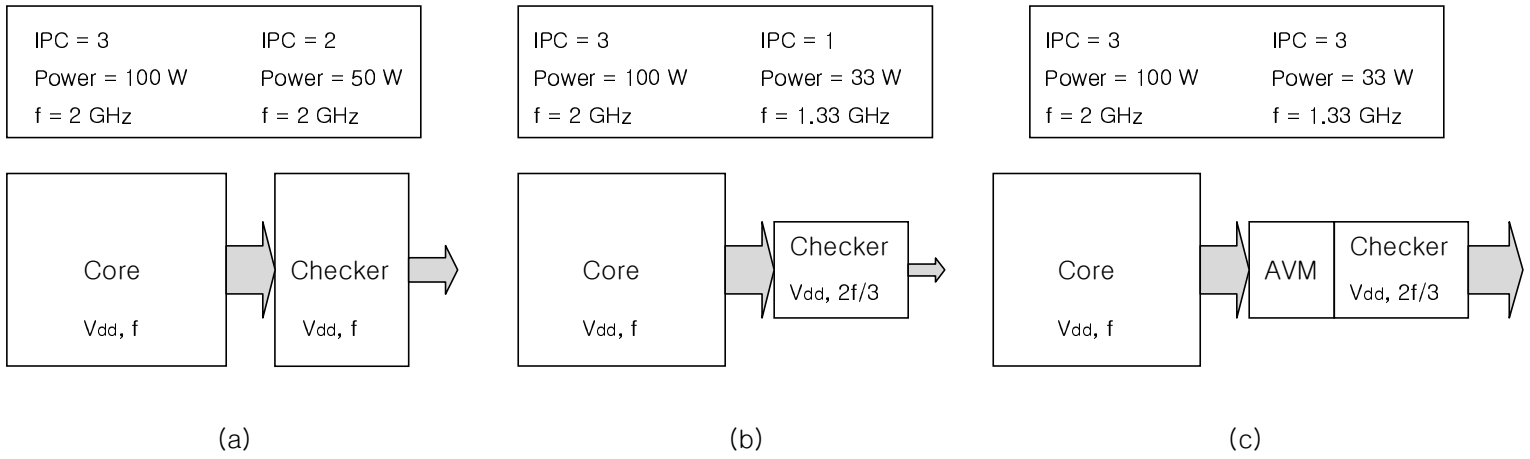


Figure 7.4: Reliable Power-efficient Architecture.

case, our AVM can be applied to distribute the reorder buffer entries more efficiently between the two threads.

In order to be readily acceptable by processor manufacturers and customers, fault tolerance schemes must satisfy three criteria—low hardware overhead, low performance overhead, and good fault coverage. Though perfect fault coverage would be required in specialized servers and mission-critical flight systems in space applications, the rest of the market does not need perfect fault coverage. Commercial desktops and commodity servers can trade off performance and reliability to arrive at reasonable design points. In such applications, employing the AVM to achieve both high performance and reasonably good reliability, makes sense. Therefore, the AVM is useful in applications where both performance and reliability are important.

Chapter 8

Related Work

Computer architects have repeatedly shown that narrow values have a strongly biased distribution, say a few narrow values account for the majority of values used. They have observed that a large percentage of the values processed in the pipeline are narrow and can be encoded with less bits than full data width [1, 17, 28, 73] or just identified [12, 25, 34]. Several value compression techniques are taken advantage of by several micro-architectural techniques, such as designing very low power pipelines [17] and cache energy reduction [1, 28, 73]. While previous studies employ value compression techniques mainly for reducing energy consumption such as register file or data cache, this chapter exploits them for improving both performance and reliability.

A host of concurrent system-level fault tolerance techniques has been proposed exploiting additional redundant execution processors [4, 44, 50, 51, 54, 68, 72]. Such redundant execution techniques employ Simultaneous Multi-Threading (SMT) [44, 51, 54, 72] or Chip Multi-Processing (CMP) [4, 68] to provide coarse-grain instruction replication with a modest amount of additional hardware support. Because the main thread shares the limited resources with the redundant thread, the performance degradation problem caused by redundant execution is common to all fully-fault-tolerant architectures. Most of previous studies have focused mainly on

the achievement of reliability with little attention on the performance overhead incurred by re-execution. The proposed work considers performance of fault-tolerant processors to explore the trade-off between performance and reliability.

Running multiple threads in thread-level redundancy solutions places a significant pressure on the processor resources, resulting in a considerable performance loss. Recently, efficient resource management techniques [30, 35, 65] between the main leading thread and the redundant trailing thread have been proposed in an SMT implementation. Exploring a partial redundancy for soft error detection is similar to opportunistic transient fault detection [30], in which the redundancy is opportunistically adjusted according to the amount of Instruction Level Parallelism (ILP). Our approach exploits a value-based correctness criticality property and manages the redundancy pro-actively based on the biased distribution of Likelihood of Correctness Criticality (LoCC).

The concept of Binary Correctness Criticality (BCC) borrows from computing Architectural Vulnerability Factor (AVF) proposed by Mukherjee *et al* [45]. While AVF is used to statically compute the vulnerability factor of a hardware structure, our correctness criticality metric is applied to dynamic instruction stream for characterizing likely correctness critical instructions. Similar likelihood function has been applied for criticality analysis of program execution latency for clustering in superscalar processors [55]. Our LoCC metric is different in that criticality in terms of correctness viewpoint is dynamically exploited to prioritize verification candidates using the filter checker.

Computer architects have repeatedly shown that data values have a strongly

biased distribution, say a few values account for the majority of values used. This value locality has been exploited by several micro-architectural techniques. Both value prediction [36] and instruction reuse [66] are such techniques, which takes advantage of identical values seen repeatedly in the pipeline. Another usage is the cache hierarchy, which works good because of the principle of address locality. They exploit either temporal or spatial locality in the same data values.

A partial value locality is defined by the repeated occurrence of partially identical data values. Some techniques exploit this property for organizing a content-aware integer register file [31], sharing physical registers [33], or information redundancy [24]. Value similarity is a special case of partial value locality. A similarity measure is usually used to cluster objects into groups with similar properties. This work characterizes it with a distance metric to improve performance and reliability of fault-tolerant processors. Some techniques exploit narrow values to optimize processor resources [12, 34]. Narrow values have a simple high-order bits with either all 0's or all 1's and do share high-order bits. Therefore, the narrow value is a subset of similar values and it is treated in a simple way in the set of similar values.

Exploring partial redundancy for soft error detection is similar to opportunistic transient fault detection [30], in which the redundancy is opportunistically adjusted according to the amount of Instruction Level Parallelism (ILP). Our approach exploits a partial value locality property and manages the redundancy pro-actively based on the biased distribution of similar values. Given the proposed fault detection is based on an anomaly speculation, it is worth to address the differences between our approach and ReStore [74]. Their symptom-based soft error detection

considers mispredictions of high confidence branches as symptoms of soft errors. Instead of using branch mispredictions as symptoms, we use an abnormal result value deviation from the nominal similarity distance as a potential soft error, exploring a partial value locality.

Chapter 9

Conclusion

9.1 Summary of Contributions and Implications of the Research

Microprocessors are becoming increasingly susceptible to soft errors due to the current trends of semiconductor technology scaling. Traditional redundant multi-threading architectures provide good fault tolerance by re-executing all the computations. However, such a full re-execution significantly increases the demand on the processor resources, resulting in severe performance degradation.

In Chapter 3, we showed that dynamic verification using the checker processor introduces severe degradation in performance unless the checker is as fast as the main processor core. Without widening the checker's bandwidth, we proposed an Active Verification Management (AVM) approach that utilizes a checker hierarchy. Based on a simplified queueing model, we evaluated the proposed AVM analytically.

In Chapter 4, before an instruction is verified at the checker processor, a binary correctness criticality based filter checker marks a Correctness Non-criticality Indicator (CNI) bit to indicate how likely its result is to be unimportant for reliability. AVM uses the CNI information to realize a congestion avoidance policy. Both reactive and proactive congestion avoidance policies are proposed to mitigate the performance degradation caused by the checker's congestion. Our experimental results show that AVM has the potential to solve the verification congestion prob-

lem when perfect fault coverage is not needed. With no AVM, congestion at the checker badly affects performance, to the tune of 57%, when compared to that of a non-fault-tolerant processor. With good marking by AVM, the performance of a reliable processor approaches 95% of that of a processor with no verification. Although instructions can be skipped on a random basis, such an approach reduces the fault coverage. A result-usage-based filter checker with a marking policy correlated with the correctness non-criticality metric, on the other hand, significantly reduces the soft error rate. We also presented results showing the trade-off between performance and reliability.

Chapter 5 further improves the AVM by designing a result-bitwidth-based filter checker, which prioritizes the verification candidates so as to selectively do verification. Binary Correctness Criticality (BCC) and Likelihood of Correctness Criticality (LoCC) are metrics that quantify whether an instruction is important for reliability or how likely an instruction is correctness-critical, respectively. A likelihood of correctness criticality is computed by a value vulnerability factor, which is defined by the numerically significant bit-width used to compute a result. The result-bitwidth-based filter checker is accomplished by exploiting information redundancy of compressing computationally useful data bits. Based on the likelihood of correctness criticality test, the filter checker mitigates the verification workload by bypassing instructions that are unimportant for correct execution. Extensive measurements prove that the LoCC metric yields quite a wide distribution of values, indicating that it has the potential to differentiate diverse degrees of correctness criticality. Experimental results show that the proposed scheme accelerates a tradi-

tional fully-fault-tolerant processor by 1.7 times, while it reduces the soft error rate to 18% of that of a non-fault-tolerant processor.

Chapter 6 presents a result-anomaly-based filter checker to guide a verification priority before the re-execution process starts. The anomaly speculation is accomplished by exploiting a value similarity property, which is defined by a frequent occurrence of partially identical values. Based on the biased distribution of similarity distance measure, we investigated further application to exploit similar values for soft error tolerance with anomaly speculation. Extensive measurements prove that the majority of instructions produce values which are different from the previous result value only in a few bits. Experimental results show that the proposed scheme accelerates the processor to be 180% faster than traditional fully-fault-tolerant processor with a minimal impact on overall soft error rate.

Chapter 7 discusses overheads of the proposed filter checker on area, hardware complexity, reliability and performance. We also discussed other potential applications of the proposed active verification management techniques.

9.2 Future Direction

The challenge for computer architects is to improve the reliability of microprocessor while maintaining its performance. My current research has addressed this problem by developing an active verification management methodology for reliable microprocessors and has investigated its performance and reliability. In this dissertation, we show that our AVM approach is very effective in achieving low

hardware overhead, low performance overhead, and good fault coverage, compared to traditional fault-tolerant techniques. We believe that the idea presented in this dissertation can be extended to wider range of problems.

Based on the success of my current research, I foresee expanding this work into several related areas and this line of research has broad relevance for other areas of computer architectures and embedded systems, including:

- Reliability in soft computing:
- Power-efficient dynamic verification:
- Model-based design and its assurance in FPGA/Firmware/Software:
- Dynamic cache coherence technology in multi-core microprocessors:
- CPU resource management technology:

Finally, while the contributions of my dissertation are mostly for CPU processors, one of projected research I expect is to connect our verification methodology with model-based design technologies, particularly in the implementation of FPGA, Firmware and Software areas. The emergence of the new generation of complex FPGAs/ASICs with embedded processors, memories, and DSP blocks reformulates the design process as well as the nature of FPGA/Firmware/Software implementation verification. Thus, applying the design for testability and the methodology for reliability concepts to FPGA/Firmware/Software are critical for achieving early and more thorough detection of faults in digital electronics and the productivity of FPGA/Firmware/Software development process.

Bibliography

[1] Carles Aliagas, Carlos Molina, Montse Garcia, Antonio Gonzalez, and Jordi Tubella. Value compression to reduce power in data caches. *Lecture Notes in Computer Science*, 2790, 2004.

[2] H. Ando. A 1.3 GHz fifth generation SPARC64 microprocessor. In *Proceedings of the IEEE International Solid-State Circuits Conference*, 2003.

[3] Todd Austin, David Blaauw, Trevor Mudge, and Kristian Flautner. Making typical silicon matter with razor. *Computer*, 37(3), March 2004.

[4] Todd M. Austin. DIVA: A reliable substrate for deep submicron microarchitecture design. In *Proceedings of the 32nd International Symposium on Microarchitecture*, pages 196–207, Nov 1999.

[5] Todd M. Austin. Diva: A dynamic approach to microprocessor verification. *Journal of Instruction-Level Parallelism*, 2, May 2000.

[6] Robert Baumann. Soft errors in commercial semiconductor technology: Overview and scaling trends. In *Tutorial Notes of the IEEE 2002 Reliability Physics Symposium*, 2002.

[7] Alfredo Benso, Stefano Di Carlo, Giorgio Di Natale, and Paolo Prinetto. A watchdog processor to detect data and control flow errors. In *Proceedings of the 9th IEEE International On-Line Testing Symposium*, Feb 2003.

[8] Dimitri Bertsekas and Robert Gallager. *Data Networks*. Prentice-Hall, Upper Saddle River, NJ, 1992.

[9] Arijit Biswas, Paul Racunas, Razvan Cheveresan, Joel Emer, Shubhendu S. Mukherjee, and Ram Rangan. Computing architectural vulnerability factors for address-based structures. *Computing Architectural Vulnerability Factors for Address-Based Structures*, 33, May 2005.

[10] Fred A. Bower, Daniel J. Sorin, and Sule Ozev. A mechanism for online diagnosis of hard faults in microprocessors. In *Proceedings of the 38th International Symposium on Microarchitecture*, Dec 2005.

[11] David Brooks. Power-aware microarchitecture: Design and modeling challenges for next-generation microprocessors. *IEEE Micro*, 20(6), 2000.

[12] David Brooks and Margaret Martonosi. Dynamically exploiting narrow width operands to improve processor power and performance. In *Proceedings of the 5th International Symposium on High Performance Computer Architecture*, Jan 1999.

[13] Doug Burger and Todd M. Austin. The SimpleScalar Tool Set, Version 2.0. CS TR 1342, University of Wisconsin-Madison, Jun 1997.

[14] J. Adam Butts and Gurindar S. Sohi. Characterizing and predicting value degree of use. In *Proceedings of the 35th International Symposium on Microarchitecture*, Dec 2002.

[15] J. Adam Butts and Gurindar S. Sohi. Dynamic dead-instruction detection and elimination. In *Proceedings of the ASPLOS'02*, Oct 2002.

[16] T. Calin, M. Nicolaidis, and R. Velazco. Upset hardened memory design for submicron cmos technology. *IEEE Transactions on Nuclear Science*, 43(6), December 1996.

[17] Ramon Canal, Antonio Gonzalez, and James E. Smith. Very low power pipelines using significance compression. In *Proceedings of the 33rd International Symposium on Microarchitecture*, Dec 2000.

[18] Jason F. Cantin, Mikko H. Lipasti, and James E. Smith. Dynamic verification of cache coherence. In *Proceedings of the Workshop on Memory Performance Issues*, Jun 2001.

[19] Saugata Chatterjee, Chris Weaver, and Todd Austin. Efficient checker processor design. In *Proceedings of the 33rd International Symposium on Microarchitecture*, Dec 2000.

[20] Junghwan Choi, Jinhwan Jeon, and Kiyoung Choi. Power minimization of functional units by partially guarded computation. In *Proceedings of ACM/IEEE International Symposium on Low Power Electronics and Design*, 2000.

[21] Seungryul Choi and Donald Yeung. Learning-based smt processor resource distribution via hill-climbing. In *Proceedings of the 33rd International Symposium on Computer Architecture*, Jun 2006.

[22] Jamison D. Collins, Dean M. Tullsen, Hong Wang, and John P. Shen. Dynamic speculative precomputation. In *Proceedings of the 34th International Symposium on Microarchitecture*, Dec 2001.

[23] Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory*. John Wiley and Sons, 1991.

[24] Martin Dimitrov and Huiyang Zhou. Locality-based information redundancy for processor reliability. In *Proceedings of the 2nd Workshop on Architectural Reliability*, Dec 2006.

[25] Oguz Ergin, Osman Unsal, Xavier Vera, and Antonio Gonzalez. Exploiting narrow values for soft error tolerance. *IEEE Computer Architecture Letters*, 5, Jul-Dec 2006.

[26] Manoj Franklin. A study of time redundant fault tolerance techniques for superscalar processors. In *Proceedings of the IEEE International Workshop on Defect and Fault Tolerance in VLSI Systems*, 1995.

[27] Manoj Franklin. Incorporating fault tolerance in superscalar processors. In *Proceedings of High Performance Computing*, 1996.

[28] Mrinmoy Ghosh, Weidong Shi, and Hsien-Hsin S. Lee. Coolpression: A hybrid significance compression technique for reducing energy in caches. In *Proceedings of IEEE Systems-on-Chip Conference*, Apr 2004.

[29] Mohamed A. Gomaa, Chad Scarbrough, T. N. Vijaykumar, and Irith Pomeranz. Transient fault recovery for chip multiprocessors. *IEEE Micro*, 23(6), Nov-Dec 2003.

[30] Mohamed A. Gomaa and T. N. Vijaykumar. Opportunistic transient-fault detection. In *Proceedings of the 32nd International Symposium on Computer Architecture*, Jun 2005.

[31] Ruben Gonzalez, Adrian Cristal, Daniel Ortega, Alexander Veidenbaum, and Mateo Valero. A content aware integer register file organization. In *Proceedings of the 31st International Symposium on Computer Architecture*, Jun 2004.

[32] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Francisco, CA, 2003.

[33] In Pyo Hong, Ha Young Jeong, and Yong Surk Lee. Physical register sharing through value similarity detection. *IEICE Transactions on Information and Systems*, E89-D, Oct 2006.

[34] Jie Hu, Shuai Wang, and Sotirios G. Ziavras. In-register duplication: Exploiting narrow-width value for improving register file reliability. In *Proceedings of the 2006 International Conference on Dependible Systems and Networks*, Jun 2006.

[35] Sumeet Kumar and Aneesh Aggarwal. Reducing resource redundancy for concurrent error detection techniques in high performance microprocessors. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture*, Feb 2006.

[36] Mikko H. Lipasti and John Paul Shen. Exceeding dataflow limit via value prediction. In *Proceedings of the 29th International Symposium on Microarchitecture*, Dec 1996.

[37] Gabriel Loh. Exploiting data-width locality to increase superscalar execution bandwidth. In *Proceedings of the 35th International Symposium on Microarchitecture*, Dec 2002.

[38] Gabriel Loh. Width prediction for reducing value predictor size and power. In *Proceedings of the 1st Workshop on Value Prediction*, Jun 2003.

[39] Aamer Mahmood and E. J. McCluskey. Concurrent error detection using watchdog processors-a survey. *IEEE Transactions on Computers*, 37, Feb 1988.

[40] Andreas Moshovos and Gurindar S. Sohi. Microarchitectural innovations: Boosting microprocessor performance beyond semiconductor technology scaling. In *Proceedings of the IEEE*, pages 1560–1575, Nov 2001.

[41] Trevor Mudge. Power: A first-class architectural design constraint. *Computer*, 34(4), April 2001.

[42] Shubhendu S. Mukherjee. New challenges in benchmarking future processors. In *Computer Architecuter Evaluation using Commerical Workloads*, Feb 2002.

[43] Shubhendu S. Mukherjee, Joel Emer, and Steven K. Reinhardt. The soft error problem: An architectural perspective. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, Feb 2005.

[44] Shubhendu S. Mukherjee, Michael Kontz, and Steven K. Reinhardt. Detailed design and evaluation of redundant multithreading alternatives. In *Proceedings of the 29th International Symposium on Computer Architecture*, Jun 2002.

[45] Shubhendu S. Mukherjee, Christopher Weaver, Joel Emer, Steven K. Reinhardt, and Todd M. Austin. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *Proceedings of the 36th International Symposium on Microarchitecture*, Dec 2003.

[46] Eugene Normand. Single event upset at ground level. *IEEE Transactions on Nuclear Science*, 43(6), December 1996.

[47] Nahmsuk Oh, Philip P. Shirvani, and Edward J. McCluskey. Error detection by duplicated instructions in super-scalar processors. *IEEE Transactions on Reliability*, 51(1), March 2002.

[48] Sanjay J. Patel. Processor-level framework for high-performance and high-dependability. In *Proceedings of the Workshop on Evaluating and Architecting Systems for Dependability*, Jun 2001.

[49] Wasiur M. Rashid, Edwin J. Tan, Michael C. Huang, and David H. Albonesi. Exploiting coarse-grain verification parallelism for power-efficient fault tolerance. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, Sep 2005.

[50] Joydeep Ray, James C. Hoe, and Babak Falsafi. Dual use of superscalar datapath for transient-fault detection and recovery. In *Proceedings of the 34th International Symposium on Microarchitecture*, Dec 2001.

[51] Steven K. Reinhardt and Shubhendu S. Mukherjee. Transient fault detection via simultaneous multithreading. In *Proceedings of the 27th International Symposium on Computer Architecture*, Jun 2000.

[52] George A. Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, and David I. August. Swift: Software implemented fault tolerance. In *Proceedings of the 3rd International Symposium on Code Generation and Optimization*, Mar 2005.

[53] George A. Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, David I. August, and Shubhendu S. Mukherjee. Design and evaluation of hybrid fault-detection systems. In *Proceedings of the 32nd International Symposium on Computer Architecture*, Jun 2005.

[54] Eric Rotenberg. AR-SMT: A microarchitectural approach to fault tolerance in microprocessors. In *Proceedings of the 29th International Symposium on Fault-Tolerant Computing*, Jun 1999.

[55] Pierre Salverda and Craig Zilles. A criticality analysis of clustering in superscalar processors. In *Proceedings of the 38th International Symposium on Microarchitecture*, Nov 2005.

[56] Ethan Schuchman and T. N. Vijaykumar. Rescue: A microarchitecture for testability and defect tolerance. In *Proceedings of the 32nd International Symposium on Computer Architecture*, Jun 2005.

[57] John P. Shen and Mikko H. Lipasti. *Modern Processor Design: Fundamentals of Superscalar Processors*. McGraw-Hill, New York, NY, 2005.

[58] John P. Shen and M. A. Schuette. Processor control flow monitoring using signatured instruction stream. *IEEE Transactions on Computers*, 36, Mar 1987.

[59] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, Sep 2001.

[60] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct 2002.

[61] P. Shivakumar, S. W. Keckler, C. R. Moore, and D. Burger. Exploiting microarchitectural redundancy for defect tolerance. In *Proceedings of the 21st International Conference on Computer Design*, Oct 2003.

[62] Kevin Skadron, Mircea R. Stan, Wei Huang, Sivakumar Velusamy, Karthik Sankaranarayanan, and David Tarjan. Temparature-aware microarchitecture.

In *Proceedings of the 30th International Symposium on Computer Architecture*, Jun 2003.

[63] T. J. Slegel. IBM's S/390 G5 microprocessor design. *IEEE Micro*, 19(2), March 1999.

[64] Jared C. Smolens, Brian T. Gold, and Jangwoo Kim. Fingerprinting: Bounding soft-error detection latency and bandwidth. In *Proceedings of the ACM ASPLOS'04*, Oct 2004.

[65] Jared C. Smolens, Jangwoo Kim, James C. Hoe, and Babak Falsafi. Efficient resource sharing in concurrent error detecting superscalar microarchitectures. In *Proceedings of the 37th International Symposium on Microarchitecture*, Dec 2004.

[66] Avinash Sodani and Gurindar S. Sohi. Understanding the differences between value prediction and instruction reuse. In *Proceedings of the 31st International Symposium on Microarchitecture*, Dec 1998.

[67] Jayanth Srinivasan, Sarita V. Adve, Pradip Bose, and Jude A. Rivers. Lifetime reliability: Toward an architectural solution. *IEEE Micro*, May-Jun 2005.

[68] Karthik Sundaramoorthy, Zach Purser, and Eric Rotenberg. Slipstream processors: Improving both performance and fault tolerance. In *Proceedings of the 33rd International Symposium on Microarchitecture*, Dec 2000.

[69] Eric Tune, Dongning Liang, Dean M. Tullsen, and Brad Calder. Dynamic prediction of critical path instructions. In *Proceedings of the 7th International Symposium on High Performance Computer Architecture*, Jan 2001.

[70] Eric Tune, Dean M. Tullsen, and Brad Calder. Quantifying instruction criticality. In *Proceedings of the 11th International Conference on Parallel Architectures and Compilation Techniques*, Sep 2002.

[71] J. S. Upadhyaya and K. K. Saluja. Watchdog processor based general rollback techniques with mutiple retries. *IEEE Transactions on Software Engineering*, Jan 1986.

[72] T. N. Vijaykumar, Irith Pomeranz, and Karl Cheng. Transient-fault recovery using simultaneous multithreading. In *Proceedings of the 29th International Symposium on Computer Architecture*, Jun 2002.

[73] Luis Villa, Michael Zhang, and Krste Asanovic. Dynamic zero compression for cache energy reduction. In *Proceedings of the 33rd International Symposium on Microarchitecture*, Dec 2000.

[74] Nicholas J. Wang and Sanjay J. Patel. Restore: Symptom based soft error detection in microprocessors. In *Proceedings of the International Conference on Dependable Systems and Networks*, Jun 2005.

[75] Nicholas J. Wang, Justin Quek, Todd M. Rafacz, and Sanjay J. Patel. Characterizing the effects of transient faults on a high-performance processor pipeline.

In *Proceedings of the International Conference on Dependable Systems and Networks*, Jun 2004.

[76] Christopher Weaver, Joel Emer, Shubhendu S. Mukherjee, and Steven K. Reinhardt. Techniques to reduce the soft error rate of a high-performance microprocessor. In *Proceedings of the 31st International Symposium on Computer Architecture*, Jun 2004.

[77] Joonhyuk Yoo and Manoj Franklin. The filter checker: An active verification management approach. In *Proceedings of the IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, Oct 2006.

[78] Joonhyuk Yoo and Manoj Franklin. Anomaly speculation based filter checker via value similarity prediction. In *Proceedings of International Conference on Information Processing*, Aug 2007.

[79] Joonhyuk Yoo and Manoj Franklin. Exploiting value similarity for soft error tolerance. In *Workshop on Unique Chips and Systems held in conjunction with IEEE International Symposium on Performance Analysis of Systems and Software*, Apr 2007.

[80] Joonhyuk Yoo and Manoj Franklin. Hierarchical verification for increasing performance in reliable processors. *Journal of Electronic Testing: Theory and Applications*, Springer, in print 2007.

[81] Joonhyuk Yoo and Manoj Franklin. Prioritizing verification via value-based correctness criticality. In *Proceedings of the IEEE International Conference on Computer Design*, Oct 2007.

[82] Wei Zhang. Computing cache vulnerability to transient errors and its implication. In *Proceedings of the 20th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, Oct 2005.

[83] Huiyang Zhou. A case for fault tolerance and performance enahncement using chip multi-processors. *IEEE Computer Architecture Letters*, 5, Jan-Jun 2006.

[84] James F. Ziegler. IBM experiments in soft fails in computer electronics (1978-1994). *IBM Journal of Research and Development*, 40(1), January 1996.