EFFICIENT SOFTWARE CHECKING FOR FAULT TOLERANCE

BY

JING YU

B.Eng., China University of Science and Technology, 2003

DISSERTATION

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science in the Graduate College of the University of Illinois at Urbana-Champaign, 2008

Urbana, Illinois

Doctoral Committee:

Professor Marc Snir, Chair
Research Assistant Professor María Jesús Garzárán, Co-Chair
Professor Josep Torrellas
Associate Professor Vikram Adve
ABSTRACT

As semiconductor technology scales into the deep submicron regime the occurrence of transient or soft errors will increase. This will require new approaches to error detection. Software checking approaches are attractive because they require little hardware modification and can be easily adjusted to fit different reliability and performance requirements. Unfortunately, software checking adds a significant performance overhead.

In order to make software checking system more attractive, this dissertation proposes three optimization techniques that reduce the overhead of software error checking approaches. The first technique uses boolean logic to identify code patterns that correspond to outcome tolerant branches. We develop a compiler algorithm that finds those patterns and removes the unnecessary replicas. In the second technique we evaluate the performance benefit obtained by removing address checks before load and stores. In addition, we evaluate the overheads that can be removed when the register file is protected in hardware. The third technique ESoftCheck composes of a set of compiler optimizations to detect and remove "non-vital" checks. ESoftCheck optimizes redundant checks, checks before loop induction variables and invariants and unnecessary checks on platforms where registers are hardware-protected with parity or ECC. ESoftCheck also provides knobs to trade reliability for performance based on the support for recovery and the degree of trustiness of the operations.

In the end, this dissertation proposes an interesting future work, which is to find
a set of variables protecting which it can most likely catch harmful errors that would cause Silent Data Corruption to the program. By ignoring non-harmful errors, the cost spent on detecting them and unnecessary rollbacks can be saved.
To my husband Yue Zhou and my parents.
Acknowledgments

This dissertation would not have been possible without the support of many people.

I especially thank my advisors Professor Marc Snir and Professor María Jesús Garzarán for the invaluable guidance, encouragement, and inspiration that they have given me over the course of my studies. They are the ones who brought me to the world of computer architecture and compilers and taught me what research is about. They have been helpful, understanding, and patient during the tough times to bring this work to fruition. Their open-mindedness and integral view on research has allowed me to explore a variety of topics pertinent to the dissertation theme. I owe them a lot of gratitude for showing me the roadmaps of research.

I would also like to thank Professor Josep Torrellas and Professor Vikram Adve for their valuable time to be on my thesis committee and for their constructive suggestions and feedbacks.

I would thank my mentors at IBM Research and Google for their help and support, especially Dr. Michael Perrone, Dr. Jim Dehnert, Dr. Carole Dulong.

I would like to express my sincere gratitude to my colleagues and friends in UIUC: Mingliang Wei, Changhao Jiang, Sara Sadeghi Baghsorkhi, Eun-Gyu Kim, Shan Lu, Hong Cheng, Jing Jiang, Bin Tan, Chih-Wei Hsu, Chun-Cheng Chen, Yi-Ting Chou, Zheng Shao, Jia Guo, Shun Wang, for their tremendous academic and personal support throughout past years.

Finally, I am deeply indebted to my husband Yue Zhou and my parents
Changhong Yu and Dongyun Weng for their love, support, and encouragement throughout my life, which helped me reach where I am today.
## TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Figures</td>
<td>x</td>
</tr>
<tr>
<td>List of Abbreviations</td>
<td>xii</td>
</tr>
<tr>
<td><strong>CHAPTER 1  Introduction</strong></td>
<td>1</td>
</tr>
<tr>
<td>1.1 Motivation</td>
<td>1</td>
</tr>
<tr>
<td>1.1.1 Why Software-based Approaches for Error Detection?</td>
<td>3</td>
</tr>
<tr>
<td>1.2 Research Goals and Contributions</td>
<td>6</td>
</tr>
<tr>
<td>1.3 Dissertation Organization</td>
<td>9</td>
</tr>
<tr>
<td><strong>CHAPTER 2  Background and Previous Work</strong></td>
<td>10</td>
</tr>
<tr>
<td>2.1 Transient Fault Problem</td>
<td>10</td>
</tr>
<tr>
<td>2.1.1 Definition of Transient Faults</td>
<td>10</td>
</tr>
<tr>
<td>2.1.2 When Transistor Size Scales Down</td>
<td>12</td>
</tr>
<tr>
<td>2.2 Previous Proposals For Transient Fault Tolerance</td>
<td>14</td>
</tr>
<tr>
<td>2.2.1 Hardware-based Approaches</td>
<td>15</td>
</tr>
<tr>
<td>2.2.2 Software-based Approaches</td>
<td>17</td>
</tr>
<tr>
<td><strong>CHAPTER 3  Baseline Software Checking and Environment Setup</strong></td>
<td>20</td>
</tr>
<tr>
<td>3.1 Baseline</td>
<td>20</td>
</tr>
<tr>
<td>3.2 Issue of Indexed Addressing</td>
<td>22</td>
</tr>
<tr>
<td>3.3 Issue of Fault Coverage for Software Checking</td>
<td>23</td>
</tr>
<tr>
<td>3.4 Environmental Setup</td>
<td>26</td>
</tr>
<tr>
<td><strong>CHAPTER 4  Use of Boolean Logic to Find Outcome Tolerant Branches</strong></td>
<td>27</td>
</tr>
<tr>
<td>4.1 Overview</td>
<td>27</td>
</tr>
<tr>
<td>4.2 Compiler Algorithm</td>
<td>30</td>
</tr>
<tr>
<td>4.2.1 Shortcut Graphs Search</td>
<td>30</td>
</tr>
<tr>
<td>4.2.2 Optimization</td>
<td>32</td>
</tr>
<tr>
<td>4.3 Evaluation</td>
<td>33</td>
</tr>
<tr>
<td>4.3.1 Performance</td>
<td>33</td>
</tr>
<tr>
<td>4.3.2 Discussion on Reliability</td>
<td>33</td>
</tr>
<tr>
<td>4.4 Conclusion</td>
<td>34</td>
</tr>
</tbody>
</table>
## LIST OF FIGURES

2.1 SER of individual circuits* .............................................. 13
2.2 SER/Chip for SRAM/latches/logic* ....................................... 14

3.1 Example of baseline software replication and checking .............. 21
3.2 Example of replicating and checking indexed addressing memory access ................................................................. 23

4.1 Eliminating replicated predicate evaluation. .............................. 28
4.2 Shortcut graphs and optimizations .......................................... 29
4.3 Constructing potential shortcut graphs ...................................... 31
4.4 Optimizing shortcut graphs. .................................................. 32
4.5 Performance speedup with boolean logic optimization compared to baseline replication ......................................................... 34

5.1 Address check removal for being covered by a later check. .......... 37
5.2 Address check removal for pointer chasing. ............................... 38
5.3 Characterization of load addresses. .......................................... 39
5.4 Performance of the different optimizations normalized against the original non-replicated code. (FullRep - Fully Replicated code, NAL - No Address checks for Loads, NALS - No Address checks for Load and Store) ................................................................. 40
5.5 Fault-detection rates break down. (O - Original non-replicated code, FR - Fully Replicated code, NAL - No Address checks for Load, NALS - No Address checks for Load and Store) ......................................................... 40
5.6 Fault-detection latency for the version where address checks before loads and stores are removed ................................................................. 45

6.1 ESoftCheck removal of redundant checks of the same register, when register file is not safe-(c) and when register file is safe-(d). 49
6.2 ESoftCheck uses data dependence to remove redundant checks of different registers. ................................................................. 50
6.3 Check removal for an induction variable. ..................................... 51
6.4 Check removal when register file is safe. ..................................... 52
6.5 ESoftcheck in the presence of checkpoints. ................................... 53
6.6 Examples of covered and non-covered checks ............................... 56
6.7 Available checks for a basic block .............................................. 59
6.8 An example of computing block transfer function $Chk_{Blk}$() ........ 61
6.9 Rules to compute Gen(B), Kill(B), and Cover(B) ....................... 62
6.10 Example of difference between ESof$Check$ and CRTR on Dependence-based checking elision. .................................................. 67
6.11 Example of applying covered-check removal for SRMT .............. 69
6.12 Characterization of static checks for each type of instruction (MemUnPolluted). ................................................................. 70
6.13 Performance of the different optimizations normalized against the original non-replicated code. ............................................. 72
6.14 Fault-detection rates break down (MemUnPolluted) ............... 77

7.1 Examples of fault-injected value traces and the corresponding golden-run value trace. The items in bold italic are found different from the golden-run. .............................................................. 89
7.2 Examples of sorting fault-injected value traces by instruction PCs, and comparing the sorted value traces to corresponding golden-run value trace. ..................................................... 92
7.3 Software Fault Mask Probability. ................................................. 97
7.4 Fault detection rate and fault detection latency with simple protectors. .......................................................... 102
## List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>SER</td>
<td>Soft Error Rate</td>
</tr>
<tr>
<td>ECC</td>
<td>Error-Correcting Code</td>
</tr>
<tr>
<td>FIT</td>
<td>Failures In Time</td>
</tr>
<tr>
<td>ALU</td>
<td>Arithmetic Logic Unit</td>
</tr>
<tr>
<td>SMT</td>
<td>Simultaneous Multi-Threading</td>
</tr>
<tr>
<td>CMP</td>
<td>Chip Multi-Processor</td>
</tr>
<tr>
<td>DBCE</td>
<td>Dependence-Based Checking Elimination</td>
</tr>
<tr>
<td>DDBCE</td>
<td>Death- and Dependence-Based Checking Elision</td>
</tr>
<tr>
<td>SRMT</td>
<td>Software-based Redundant Multi-Threading</td>
</tr>
<tr>
<td>SVQ</td>
<td>Store Value Queue</td>
</tr>
<tr>
<td>LVQ</td>
<td>Load Value Queue</td>
</tr>
<tr>
<td>SEU</td>
<td>Single Event Upset</td>
</tr>
<tr>
<td>CSE</td>
<td>Common Sub-expression Elimination</td>
</tr>
<tr>
<td>ACE</td>
<td>Architectural Correct Execution</td>
</tr>
</tbody>
</table>
CHAPTER 1

Introduction

1.1 Motivation

Dramatic increases in the number of transistors that can be integrated on a chip will deliver great performance gains. However, it will also expose a major roadblock, namely the poor reliability of the hardware. Indeed, in the near-future environment of low power, low voltage, relatively high frequency, and very small feature size, processors will be more susceptible to transient faults.

Transient Faults, also known as soft errors, are due to impacts from neutron and alpha particles or other random events such as internal noise on power supply and interconnections, electromagnetic interference and electrostatic discharge, which change the logic values of latches or logic structures [11,12,18,20,60]. These changes are temporary - they do not permanently damage the hardware. However, they may result in the wrong program output: computing with wrong operands, fetching data from a wrong address, following a wrong path, etc, may finally cause the pragram to fail. Sometimes users may easily notice these faults if the program presents abnormal behaviors; sometimes, however, the program will produce the wrong output and the users will not know about it. The latter situation is more harmful. It has caused costly failures in high-end systems in recent years. For example, in 2000 Sun Microsystems reported that cosmic rays interfered with cache memories and crashed server systems at major customer sites, including America Online, eBay [4]. In 2005, Hewlett Packard stated that on a 2048-node
supercomputer in Los Alamos National Laboratory, a higher-than-expected number of single-node failures were observed and the primary cause of these failures were transient faults induced by cosmic ray strikes [32].

The continued evolution of hardware toward smaller feature size, lower voltage, and higher frequency suggests that soft error rates (SERs) will increase in the future and transient faults will become a greater problem for future generation of processors.

In fact, most modern microprocessors already incorporate mechanisms, such as parity and error-correcting codes (ECC) for detecting or recovering transient faults in memory elements. However it is not enough to protect only memory elements from transient faults. Recent studies [60] show that, by 2011 the soft error rate in combinational logic will be comparable to that of unprotected memory elements (see Section 2.1.2 for a detailed discussion on the trend of SER rate for SRAM/latches/logic). For processors that use parity or ECC to protect a large portion of the memory elements on the chip, logic will quickly become the dominant source of soft errors. This dissertation focuses on protecting the entire processor against transient faults.

Error detection mechanisms are necessary to ensure that a soft error does not go undetected and results in an erroneous computation. Once errors are detected, it is often possible to use software schemes for error correction – the performance of error correction schemes is not critical, as long as errors are not too frequent; however, error detection adds an overhead to all computations and has to perform efficiently. For this reason, we focus in this dissertation on error detection.
1.1.1 Why Software-based Approaches for Error Detection?

Hardware-based error detection is used on modern microprocessors to detect errors in storage and buses. For example, for memory elements, redundant bits are padded into each data cell in the form of parity or error-correcting codes (ECC). Whenever the cell is updated, the checking bits are re-calculated based on the new data; whenever the cell is accessed, the data is checked by the checking bits. In this way, the storage structures are able to detect or recover from transient faults occurred since the last update (or access). Generally, information redundancy (e.g., parity, ECC) adds a low overhead to performance and chip size. However, it is much harder to use the method of information redundancy in the pipeline. For example, the parity or ECC property will not hold when the data go through an Arithmetic Logic Unit (ALU).

In order to protect the pipeline, in the early 1990s, processor designers put in-line checking circuits to protect every single logical unit, such as using parity predictor to protect transformation logic and sequential controls, using residue checking to protect arithmetic functional units, etc [10]. However, in-line checking circuits need a lot of white space and requires high verification cost. This kind of fine-grain fault detection strategy is also a nightmare when processor designers want to make any small changes to the chip. Later when transistor technology became off-the-shelf, processor designers chose to protect the pipeline at a higher lever - the module level, such as IBM mainframes [61], HP NonStop [29] and mission-critical computers [75, 76], etc. These systems duplicate or triplicate their pipeline or processor core. When there are results coming out of a pipeline stage or the processor core, the out-going results are compared (results from the two identical pipelines or processor cores) to detect errors. Comparing to logic-level checking, module-level checking has a little bit longer error detection latency, but
it is much easier for the processor designer to design and verify the new checking logic.

However, for commodity processors which are cost- and energy-constrained, the technology used in these extreme reliable systems may not be suitable. This mass-market needs a fault tolerant technology that is cheap, lightweight and able to cover the entire processor efficiently. There have been several proposals targeting this goal that introduce redundancy to the execution. Depending on whether the redundancy is implemented in hardware or software, these proposals are classified into hardware-based approaches and software-based approaches (see Section 2.2 for detailed discussion). Hardware approaches include auxiliary simple checker processors [3, 9], redundant hardware threads [16, 35, 52, 67] and others. Software-based approaches use the compiler to introduce redundant instructions to the program and add checks before stores and other synchronization instructions to ensure that the execution is correct [42, 53, 54, 56, 68].

However, it is unclear whether such hardware cost is acceptable for commodity systems. For such systems, software-based error detection may be a preferable solution. Software-based checking approaches have the following advantages.

- **Little Hardware Cost:** Software-based checking approaches need little hardware modification. Therefore, it does not have much extra cost on circuit design and verification. In addition, when processors are upgraded with new techniques, the checking system do not have to be trashed and designed from scratch.

- **Portable:** Since the redundant computations are introduced by the compiler in the software-based checking approaches, if the compiler has support for several platforms, then this checking system can be ported to other platforms with just small changes.
• **Flexible:** The main advantage of software-based checking approaches is their flexibility: different trade-offs between performance and reliability can be achieved on the same hardware, using different software approaches while hardware-only solutions cannot offer the same flexibility. Such flexibility can be used, for example, for achieving a higher level of reliability for large clusters, built out of commodity components: a PC might be built to have a mean time between undetected failures (MTTUF) of, say, 10 years; this would result in an unacceptable MTTUF of half a week for a 1000-node PC cluster. Alternatively, the flexibility may be used to achieve different levels of reliability for different software components: one may not care about undetected errors that will affect the PC display during a game, but may want to avoid errors that will corrupt the file system metadata.

Unfortunately, software-based checking approaches add a significant performance overhead. On Itanium, the fully protected code performs 1.41 times slow as the original out-of-box code [53]; on X86 platforms where the register spill is a big issue, the protection overhead is much higher: our experiments show that the baseline fully protected code runs 2.16 times slow as the original code version (see Section 6.4.2), and Reis et. al show a 3.22 times slow in their experiments [55]. Notice that, there is no apple-to-apple comparison between the numbers above, because i) the above experiments are done using different infrastructures, ii) redundant instructions are added at different compiler levels, iii) they make different assumptions on possible faults. Numbers can only be compared when they are obtained using the same experimental setup.

Although software-based checking approaches have some performance concerns, considering their advantages, we still believe they have great potentials to detect transient faults for commodity processors. In order to make software-based checking approaches more attractive, this dissertation mainly studies the
optimization opportunities in the software-based checking approaches with the goal of reducing the performance overhead down to an acceptable level and at the same time keep the same or similar level of reliability.

1.2 Research Goals and Contributions

As stated in Section 1.1.1, software-based checking approaches have great potentials to detect transient faults for modern commodity processors because of their advantages in cost and flexibility, though there are concerns on their performance overhead. In order to make software-based checking approaches more attractive, this dissertation focuses on optimization techniques that reduce the performance overheads of software-based fault detection approaches, while taking into account the following issues:

- **Reliability** Detecting transient faults is still the only functionality of a fault detection framework. When working on reducing the performance overheads, we need to think whether the optimization techniques affect the fault detection capability and if so, how much is the impact. Furthermore, since software-based checking approaches assume a software/system checkpoint and recovery framework, it has to be guaranteed that the underneath checkpoint recovery layer is always in a healthy state, meaning that when an error is detected the checkpoint recovery framework is able to roll-back and recover correctly. In this dissertation, “reliability” means fault coverage and system recovery capability.

- **Flexibility** Flexibility is the main advantage of software-based checking approaches when compared to hardware-based approaches. To maintain this advantage our optimization techniques need to be adjustable to fit different
reliability requirements. When the reliability requirement is high, the optimization will be conservative; when the reliability requirement is low, the optimization can be more aggressive; in the limit, when fault protection is not needed at all, the whole software-based checking transformation can be disabled.

For this work we consider a baseline software-based approach that replicates computing instructions and adds checks before stores and other synchronization instructions to ensure that data stored in memory are correct [53, 54, 56, 68]. With this approach, the checking instructions account for a significant fraction of the added overhead. For example, Reis et al [55] found that on average 46% of the added overhead is due to the checking instructions. To reduce the overheads and improve performance it is necessary to reduce the number of replicas and/or checking instruction. The contribution of this dissertation is the proposal of several optimization techniques to reduce overheads. These techniques include:

- **The Use of Boolean Logic to Find Outcome Tolerant Branches**

  The Boolean Logic Optimization is based on the fact that programs already have redundancy, and if the compiler can determine the programs sections where such redundancy exists, it can avoid the replication and later checking. We use boolean logic to identify a code pattern that corresponds to outcome tolerant branches and develop a compiler algorithm that automatically finds those patterns and removes the unnecessary replicas.

- **The Removal of Address Checks**

  This technique is based on the observation that faults that corrupt the application tend to quickly generate other noisy errors such as segmentation faults [70]. Thus, we can reduce replication of the instructions that tend to
generate these type of errors, trading reliability for performance. In this study, we remove the checks of the memory addresses and discuss situations where removing these checks affect little to the fault coverage. This occurs when a check of a variable is covered by a later check to the same variable, and thus errors in the first check will be detected by the later checks; and in pointer-chasing, when the data loaded by a load is used immediately by another load.

- **ESoftCheck: Removal of “non-vital” Checks**

ESoftCheck contains a set of novel compiler optimizations that identify and remove the redundant “non-vital” checking instructions while keeping a level of reliability similar to the one of the non-optimized code. ESoftCheck applies three classes of optimizations: i) it removes redundant checks. A check of a variable, say $v$, is redundant if it dominates another nearby check of $v$ or of a variable whose value is a function of $v$; ii) it hoists out of loop checks of loop-invariant or induction variables. iii) it removes checks that are unnecessary when the register file is hardware-protected with parity or ECC. Machines with hardware-protected register files include Intel Itanium [30], Sun UltraSPARC [21] and IBM Power4-6 [5]. An additional benefit of this last optimization is that the register holding the replicated register can be deallocated earlier, reducing register pressure. As a flexible software-based approach, ESoftCheck also provides knobs so that the user can trade reliability for performance. With ESoftCheck the user can i) define what are the commit points, and ii) specify the degree of trustiness of each operation. The distance between commit points is important, because the larger the number of instructions between commit points, the more redundant checks can be detected and removed. The degree of trustiness affects the removal
of checks on variables that are data dependent through trusted operations.

1.3 Dissertation Organization

This dissertation is organized as follows. Chapter 2 introduces the background of transient fault errors and the related work on transient fault tolerance. Chapter 3 describes the baseline software-based fault detection system and our environmental setup. Chapter 4 presents a technique to detect outcome tolerant branches and its performance benefit. Chapter 5 describes the removal of address checks and evaluates its impact in performance. Chapter 6 presents an overview of ESoftCheck, the detailed compiler algorithms that ESoftCheck uses and the evaluation results. Chapter 7 proposes future work on finding SDC-indicating variables and designing an intelligent fault detector that can most likely to catch only harmful errors. And finally, Chapter 8 concludes this dissertation.
CHAPTER 2

Background and Previous Work

In this chapter, we first describe in detail the problem of transient faults and the trend of transient faults problem as the processors scale down into the deep submicron regime; then, we present previous works on tolerating transient faults for commodity processors.

2.1 Transient Fault Problem

This section introduces the nature of transient faults. Section 2.1.1 presents the definition of transient faults. Section 2.1.2 discusses the trend of transient fault as the semiconductor technology scales down and addresses the necessity of protecting combinational logic from transient faults.

2.1.1 Definition of Transient Faults

Transient Faults, also known as soft errors, are due to impacts from energy particles or other random events such as internal noise on power supply and interconnections, electromagnetic interference and electrostatic discharge, which change the logic values of latches or logic structures [11, 12, 18, 20, 60]. These changes are temporary - do not permanently damage the hardware. However, they may result in a program to run incorrectly by: computing with wrong operands, fetching data from a wrong address, following a wrong path, etc, and finally cause the program to fail. Sometimes users may easily notice these faults if the program
presents abnormal behaviors; sometimes users will not know until the program ends and produces a wrong output.

There are two sources for the energy particles: the decay of radioactive atoms which exist in trace amounts in all materials and extraterrestrial cosmic rays which bombard the earth constantly from the far depths of the galaxy. The alpha-particle is proved to be the most upsetting ones in the decay of radioactive atoms. When the 16Kb chip was introduced in 1977, the storage charge in memory cells had been reduced to about 1M electrons, which is just the amount that an alpha-particle can cause a sudden burst. In 1978, it was the first time that Intel discovered that radioactive contaminants in the packaging of chips caused changes in the memory elements of a processor [28].

The transient faults caused by cosmic rays was first discovered in satellites in 1950s. Solutions included covering the satellite with a gold “blanket” and encoding the data streams with parity and handshaking during transmission. [77] Since cosmic rays can penetrate the atmosphere, cosmic rays still play an important role in causing transient faults on the earth. Because the cosmic ray flux increases in intensity with altitude, the altitude of the processor has a significant effect on the SER rate. A field test from IBM showed that SER increases more than 10x going from sea level to two miles up [38]. If using New York City at sea level as a baseline, the SER rate increases 4x when moving up to Denver, Colorado and increases to 13x at Leadville, Colorado. At the altitude of airplane travel the SER rate increases further to 300x. Decreasing altitude below sea level has the inverse effect. For example, moving 20 meters underground reduces the SER rate to 3% of sea level [1]. Although the layer of covering or putting computers underground can help to shield cosmic rays and limit transient faults, it is too expensive and impractical for use widely.
2.1.2 When Transistor Size Scales Down

The continued evolution of hardware toward smaller feature size, lower voltage, and higher frequency suggests that soft error rates (SERs) will increase in the future. On one hand, shrinking transistor size decreases the probability of collecting the critical charge that is necessary to upset a circuit. On the other hand, the critical charge itself decreases even faster, because of lower cell capacity and supply voltage, leading to higher soft error rates (SERs). In the future, with technology advancing both transistor size and critical charge will continue to decrease. Most of the neutron and alpha strikes will be able to flip a bit within a cell because of the very low critical charge. The probability of corrupting data in a particular bit cell will decrease because of the lower area. However, the SER per chip will continue to increase, because of the use of larger arrays [12].

Since it is impractical to place all processors underground to shield cosmic radiation and it is almost impossible to insulate circuits from alpha particles coming from packaging materials, transient faults will become a greater problem for future generation of processors.

Most early studies focused on protecting memory elements from impacts of transient faults. The main reason is that caches take up a large part of the chip area. Thus the probability of energy particles striking a memory cell is higher. As a result, most modern microprocessors already incorporate mechanisms, such as parity and error-correcting codes (ECC) for detecting or recovering transient faults in memory elements. However it is not enough to protect only memory elements from transient faults.

Recent studies [60] show that, by 2011 the soft error rate in combinational logic will be comparable to that of unprotected memory elements. Figure 2.1 shows the predicted SER for each class of components for each technology generation.
Figure 2.1: SER of individual circuits* and pipeline depth studied by Shivakumar et al. [60]. The x-axis plots the CMOS technology generation, arranged by actual or expected year of adoption, and the y-axis plots the SER for each element on a log scale. The SER is expressed in terms of Failures In Time (FIT), which measures the number of failures per $10^9$ hours of operation. On this figure, SER of a single SRAM cell decreases gradually with shrinking feature size, while the SER of a latch stays relatively constant. The SER for a single logic chain shows the most significant change, increasing over five orders of magnitude from 600nm to 50nm.

If the Alpha 21264 microprocessor is used as the basis for constructing the chip model, where approximately 20% of transistors are in logic circuits and the remaining 80% are in storage elements in the form of latches, caches, branch predictors, and other memory structures, SER/chip can be computed and presented in Figure 2.2. SER/chip of SRAM remains almost flat as feature size decrease. SER/chip of latches increases only slightly for all pipeline depths. Similar to

\footnote{Figure 2.1 and Figure 2.2 are taken from Reference [60].}
Figure 2.1: SER/chip in combinational logic increases dramatically from $10^{-7}$ to around $10^2$ as the technology evolves from 600nm to 50nm. We can also see that at 50nm with 6 fan-out-of-4 inverter (FO4) pipeline, the SER/chip of combinational logic exceeds that of latches, and is within two orders of magnitude of the SER/chip of unprotected SRAMs.

For processors that use parity or ECC to protect a large portion of the memory elements on the chip, logic will quickly become the dominant source of soft errors.

2.2 Previous Proposals For Transient Fault Tolerance

As transient fault problem is rising for commodity processors, there has been a rich body of proposals on transient fault protection schemes. Here we focus on some of the key schemes using a hardware or a software approach.
2.2.1 Hardware-based Approaches

Watchdog Processor Concept Watchdog processors proposes some of the key concepts of many fault-tolerance schemes [27]. A watch dog processor is a processor that runs concurrently with the main processor, observes the main processor’s outputs and inputs, and compares its own outputs with the main processor’s outputs.

Redundant Multi-Threading Along with the recent popularity of SMT (Simultaneous Multi-Threading) and CMP (Chip Multi-Processor) processors, there have been several proposals using redundant threads to check the correctness of executions, on the same core (SMT) or on different cores (CMP). The basic idea is very simple: run two copies of the same thread and check with each other. Depending on the platforms (SMT vs. CMP) and the functionality (detection and recovery), there are various proposals.

AR-SMT [57] is the first to use SMT to execute two copies of the same program. Two threads are checked before an instruction commits at the trailing thread side. The committed state of the trailing thread can be used as a checkpoint for recovery. Slipstream [65] follows AR-SMT by extending the scheme to CMP. AR-SMT and Slipstream propose using speculation techniques to allow communication of data values and branch outcomes between the main the redundant threads to accelerate execution. AR-SMT doubles the physical memory of a system: loads and stores are performed twice by the two threads. Thus it requires operating system modifications to manage the additional address mappings needed to replicate the address space. SRT [52] introduces the concept of the sphere of replication; inputs and outputs values that cross the boundary of the sphere require replication and comparison respectively. If memory is outside sphere of replication, the leading thread needs to replicate loaded values and copy these values to the trailing thread;
only committed stores and uncached loads need to be checked. Another optimization of SRT is to maintain a constant slack of instructions between the two threads. SRTR [67] extends SRT to provide recovery for SMT. It checks more instructions to make sure the architectural states committed by the trailing thread are always correct and thus can be used to recover. CRT [35] exploits design options for CMP. It proposes to replace the branch outcome queue with a little prediction queue to help replicate the instruction stream in the trailing thread, and proposes a per-thread store queue to enhance performance. CRTR [16] extends CRT to provide recovery for CMP. To tackle inter-processor bandwidth, CRTR increases the bandwidth supply by pipelining the communication paths and reduces the number of checks by using an advanced dependence-based checking elision over SRTR. Fingerprinting [62] further reduces inter-processor bandwidth. It collects the history of the execution of the processor in a hash-based signature and uses them to detect differences among dual modular redundant pair of processors.

There are other kinds of instruction-level replication on platforms other than SMT and CMP. DIE proposes hardware recovery using superscalar hardware without any SMT support [50]. DIVA [3,9] uses a very simple in-order processor as a checker for a large out-of-order, speculative processor. DIVA’s checker is a completely separate processor. It is assumed always safe. As a result, DIVA’s checker can detect permanent faults and design errors in the main processor as well as transient faults.

Partial Redundant Threading Recently, several lightweight hardware redundant thread approaches have been proposed to duplicate only a subset of the dynamic instruction streams at the cost of possibly lower error detection and recovery rate. The idea behind these approaches is to improve the overall cost-effectiveness to make reliable computing practical. Weaver et al. [73] proposes two techniques to reduce the cost: flush certain pipeline structures on infrequent long
latency events, such as cache misses; modify the error reporting mechanism and only signal dangerous detected errors that could possibly affect instructions. Gomaa et al. [17] proposes an opportunistic scheme: use implicit redundancy through reuse in high-ILP phases and explicit redundancy in low-ILP phases. DIE-IRB [44] exploits instruction reuse to reduce redundant execution. Soundararajan et al. [64] monitors the upper bound of the architectural vulnerability of Reorder Buffer (ROB) online and provides control mechanisms to proactively bound the vulnerabilities to any limit specified by the system designer.

**Prediction-Based Checking** Prediction-based checking is a new direction in efficient fault tolerance. Since errors may produce symptoms in the system or may violate some localities, these symptoms or violations can be used as signs of the appearance of soft errors. Wang et al. proposes ReStore [70], in which fault detection is achieved purely by symptoms such as exceptions, cache misses, TLB misses, and branch mispredictions. Li et al. [23] observes that most hardware permanent faults cause anomalous behaviors as fatal hardware traps, segmentation fault, operating system hangs and high operating system activity, and proposes to use these symptoms to find possible hardware faults. Reddy et al. [51] combines confident predictions and partial duplication and shows that the combination can approximate the fault tolerance of full duplication. Parashar et al. [45] uses the value and control-flow locality to predict instruction results; for the instructions whose results can not be predicted, it extracts their backward slices and redundantly execute these slices.

### 2.2.2 Software-based Approaches

Software-based approaches basically adds replicated instructions at compile time; checks are inserted to compare the two copies of the execution streams. Oh et al.
proposes EDDI [42], a software system where all the instructions are replicated and interleaved with the original ones. Checking instructions are inserted at synchronization points and memory usage is doubled. Oh et al. later extends EDDI by proposing ED4I that is able to detect some permanent errors [41]. In ED4I, the replicated instructions “shift” the computation operands such that the original and replicated streams may operate on different hardware units. Reis et al. proposes SWIFT [53] and Profit [56] that improves over EDDI by taking advantage of the current protection of the memory subsystem so that memory is not replicated. They also add control flow checking. J. Chang et al. [7] uses transformed replicas to recover errors in integer arithmetic operations and integer registers. Wang et al. [68] is the first to apply software-based checking schemes to multicore processors, called Software-Based Redundant Multi-Threading (SRMT). The compiler-generated replicated instructions and checking instructions are executed on another core. SRMT defines two non-repeatable operations: system calls for I/O operations and shared memory access operations. For non-repeatable operations, their return values need to be duplicated and their operands need to be checked at the trailing thread side. To give the compiler more freedom for possible optimizations, the leading thread does not wait for the acknowledgement from the trailing thread, except in situations where an error may have adverse side effect to the outside world.

Some previous works provide ways to trade fault coverage for performance or power consumption. The work by Oh and McCluskey [39] trades error detection latency for power consumption by replicating selected function calls instead of replicating all instructions of each function. However, this technique may not benefit performance and it can only be applied to functions that do not update memory or global variables. PROFiT [56] and Spot [55] select some sections of the program to replicate. The selection is done based on a profile of the execution
time and on the fault injection results for each section.

Hybrid software checking systems are proposed to mitigate the high cost in pure software checking systems [47,54]. These systems use several special designed hardware structures that were originally proposed in hardware redundant multi-threading schemes. For example, a load value queue is used to check if the two loads issued by the original and replicated instruction stream are requesting the same address and to make sure that these two loads get the same loaded value. A store value queue is used to hold the stored value until two stores arrive and check if the two stores are the same. With these hardware support, checks will be done by the hardware greatly reduce the performance overhead. SRMT [68] also evaluates their designs in two environment: one with a hardware value queue, the other with a software queue. The design with a software queue is about 2.4x slow down compared to the one with a hardware queue.

Compiler techniques have been used in other approaches for fault tolerance. Meixner et al. [31] compute dataflow graphs at compiler time and use special hardware to verify the dataflow when the program is executing. Nakka et al. [37] select "critical" variables through data dependence analysis, and use a hardware redundant thread to check the slices that contribute to these variables.
CHAPTER 3

Baseline Software Checking and Environment Setup

In this chapter, we first describe the baseline software-based error checking model and discuss emerging issues on addressing mode and fault coverage. Then, we present our implementation method and environment setup.

3.1 Baseline

Software techniques for fault tolerance such as SWIFT [53, 56] assume that data can be corrupted in arbitrary ways within the CPU but that memory and caches are error-free; i.e., that the protection offered by techniques such as ECC and memory scrubbing reduce the frequency of undetected errors to an acceptable level. The software techniques detect CPU errors and prevent a faulty value from being written to memory. The base approach for doing so is to keep two copies of each register value, and to execute each operation twice, on different copies of the data; errors are detected by comparing the two copies. Stores, branches, function calls, returns, and loads are considered to be “synchronization” points where we need to ensure that certain values are correct; checking instructions are inserted before each synchronization point:

- Before a store, checking instructions verify the value and memory address. This ensures that the correct data is stored to the correct memory location.

- After a branch, checking instructions verify that the branch takes the appropriate path.
Before a function call, checking instructions verify the input parameters.

Before a function return, checking instructions verify the return value.

Before a load, checking instructions verify the address of the load. Then, the loaded value is immediately copied to another register [7].

An example of the original and its corresponding augmented code executing in the same thread is shown in Figure 3.1-(a) and (b), respectively. The augmented code contains additional instructions that are shown in bold and uses additional registers that are marked with a ’. Instruction 1 and 5 replicate the additions, instructions 2 and 3 check that the load is loading from the correct address, instruction 4 copies the value just loaded in r3 and instruction 6-9 check that the store writes the correct data to the correct memory address.

Figure 3.1: Example of baseline software replication and checking

Figure 3.1-(c) and (d) show an example of replicating the conditional branch, where Figure 3.1-(c) is the original assembly code, and (d) is the augmented code. Instruction 1 and 2 replicate the comparison and check if it is correct for the control flow to take the not-taken branch; instruction 3 and 4 replicate and
check the comparison on the taken branch. However, this type of checking is not
enough if the control flow jumps illegally. We will discuss this issue in Section 3.3.

3.2 Issue of Indexed Addressing

In the example of Figure 3.1 loads and stores use indirect addressing, so that the
register arguments contain the address of the memory location accessed. A slight
issue arises with indexed addressing, where the register specifies an offset, not the
address itself. An example is shown in Figure 3.2, where the original code is shown
in Figure 3.2-(a). In (ld r3 = c1[r1]), the address is the contents of the register
r1 plus the constant c1 that is part of the instruction. Our baseline replicated
code, shown in Figure 3.2-(b), will check the contents of the register (cmp r1,
r1'), as in the indirect addressing case. However, the test for r1 leaves open a
window of vulnerability in the address adder: an error in the implicit addition will
not be detected. Protecting the implicit addition is possible by replacing it with
an explicit addition and using indirect addressing, as shown in Figure 3.2-(c). The
addition is explicitly computed by instruction 1 and replicated by instruction 2.
Instructions 3 and 4 check if the addition is correct.

Although the version (c) may seem more reliable, errors are still possible: there
is a window of vulnerability between the check of the register value and the use
of this value. Also, the adder used in the address unit is simpler than the integer
ALU, and likely more reliable; it is not clear that the version (c) is significantly
more reliable than the version (b). In addition, on X86 platforms where the
indexed addressing mode has been massively used and optimized, version (b) will
be more efficient than version (c). Therefore, for the experiments carried out
in this dissertation, the replicated code in Figure 3.2-(b) is considered to be the
baseline.
Figure 3.2: Example of replicating and checking indexed addressing memory access

### 3.3 Issue of Fault Coverage for Software Checking

Software-based instruction level replication can detect errors happening not only to computational unit, register files but also to varied buses, combinational units where the errors would propagate to the resulting value of an instruction. However, software-only approaches can not provide 100% fault coverage. In our baseline and improved model, the following transient faults may not be detected.

I Strictly speaking, software checking is not fully reliable. Because an error could happen after the check and before the store. For example, if an error occurs after the check for store address and store value is done, and before the store instruction executed, the error will propagate to memory without being detected. However, software checking approaches reduce the window of vulnerability where a soft error can cause a faulty value to be written to memory.

II On a load, although memory is error-free it is still possible for an error to occur in the path that brings the data from memory to the register file. Replicating load instructions and loading from the same address twice would be a solution. But in multi-thread programming environment, loading from the same address twice can result in different loaded values, which has to
be forbidden. A hardware Load Value Queue (LVQ) can better solve this problem at extra hardware cost [35, 52, 54]. For the same reason, a store path from register to memory is vulnerable to transient errors and that the error may make the store value or store address wrong. A hardware Store Value Queue (SVQ) can solve this problem with extra hardware cost.

III With the approach described in Section 3.1 branches are checked to verify that they follow the appropriate path, but the program could follow an illegal control path if the program counter is corrupted. Thus, additional testing for legal control flow, done either in software or in hardware, is necessary to ensure that the program counter is not corrupted [2, 25, 27, 40, 43, 66].

IV For function calls, our baseline replication checks the function arguments before calls the function in the usual way. It is possible that the argument is corrupted in the middle of the transfer. SWIFT [53] discusses this type of vulnerability and proposes a way to mitigate the error: doubling the size of a function arguments list, giving two copies of each argument to the callee and checking the arguments right entering the callee function.

V The software-based instruction level replication can detect most transient errors to instruction op code, if the wrong-interpretated instruction generates a value that is different from the correct instruction. But, if an error changes a non-store instruction to a store instruction, the error may propagate to memory before it is caught; or if an error changes a store instruction op to a null op, the store will not be executed and the error will not be detected; etc. A hardware Store Value Queue can mitigate these errors.

VI If an error cause an exception that should not happen in correct runs, there are two cases: if the exception handler finally returns, the error may be
detected by the software inserted checkings; if the exception handler does not return, the error will not be caught by inserted checkings unless we also add checking instructions to the exception handler. An issue rises if the exception handler makes changes to outside world (e.g. I/O), the damage is hard to recover even the error is caught by inserted checkings.

VII If software-based checking mechanism is implemented at source or intermediate level (we will explain why intermediate level is a reasonable choice in next section), the libraries that are not available in source code can not be protected in this way. There are two cases to consider. If the library call does not make external changes, we can treat this library call as a huge instruction and replicate this instruction. Otherwise, we just leave the library function open the vulnerability or protect these libraries using binary level instrumentation as presented in [55].

VIII An error in micro-architecture may manifest in multi-bit error or multiple errors in the architectural variables. Though our fault model aims at single error upset (SEU), it is still likely that multiple errors will generate unmatched pairs of variables and thus be caught by the software checkings.

In short, the software-based instruction level replication approach can not provide 100% fault coverage (though some errors can be mitigated by special hardware support or more expensive software protection). We still think this kind of non-perfect fault coverage is acceptable for commodity processor where its performance and power consumption is equally if not more, important. Based on this, we can sacrifice reliability a little bit for gaining much performance improvement.
3.4 Environmental Setup

We use LLVM [22] as our compiler infrastructure to generate single-threaded codes extended with redundant computations and the check operations. This extension is carried out at the intermediate level, right after all the static optimizations have been done. We replicate all the integer and floating point instructions. Previous implementations have added redundancy at the backend, right before register allocation [53,54] or via dynamic binary translation [55]. The advantages of working at the intermediate level are:

I The redundant code can be easily ported to other platforms and other languages.

II We do not need to fully understand the assembly code for that platform.

III At the intermediate level we see a simple memory access model rather than complex instructions like the ones in the x86 ISA.

To prevent backend optimizations to eliminate the added code we tag the replicated instructions. The backend optimizations are applied separately to the tagged and the untagged instructions. For the evaluation reported in this dissertation we use SPEC CINT2000 and the C codes from SPEC CFP2000, running with the reference inputs. Experiments are done on a 3.6GHz INTEL Pentium 4 with 2GB of RAM running RedHat9 Linux.
CHAPTER 4

Use of Boolean Logic to Find Outcome Tolerant Branches

In this chapter we explain how to use boolean logic to reduce the amount of replicated instructions. This technique is based on the fact that programs already have redundancy, and if the compiler can determine the programs sections where such redundancy exists, it can avoid the replication and later checking. We use boolean logic to identify a code pattern that corresponds to outcome tolerant branches and develop a compiler algorithm that automatically finds those patterns and removes the unnecessary replicas.

We first do an overview (Section 4.1) and then explain the compiler algorithm (Section 4.2), and finally presents the evaluation result and a short conclusion.

4.1 Overview

Our technique is based on the fact that programs have redundancy. For instance, Wang et al. [69] performed fault injection experiments and found that about 40% of all the dynamic conditional branches are outcome tolerant. These are branches that, despite an error, converge to the correct point of execution. These branches are outcome-tolerant due to redundancies introduced by the compiler or the programmer. An example of outcome-tolerant branch appears in a structure such as if (A || B || C) then X else Y. In this case if A is erroneously computed to be true, but B or C are actually true, this branch is outcome tolerant, since the code converges to the correct path. The control flow graph of this structure is
shown in Figure 4.1-(a).

The state-of-the-art approach to check for errors is to replicate branches as shown in Figure 4.1-(b), where the circles correspond to the branch replicas. However, we can reduce overheads by removing the comparison replica when the branch correctly branches to X. If the original comparison in A is true we need to execute the comparison replica to verify that the code correctly branches to X. However, if A is false, we can skip the execution of the A replica and move to check B. We will only need to execute the A replica if both B and C are also false. The resulting control flow graph is shown in Figure 4.1-(c). In situations where A and B are false, but C is true, we can save a few comparisons.

![Figure 4.1](a) Original (b) State of the art (c) Optimized

Figure 4.1: Eliminating replicated predicate evaluation.

Outcome tolerant branches also appear in code structures such as if (A & B & C) then X else Y, and in general in all the code structures that contain one or more shortcut paths in the control flow graph. A basic shortcut path is edge(A->X) in Figure 4.2-(a), where both A and its child point to the same block. However, most shortcut paths are more complex. For instance, in Figure 4.2-(b), block A points to the same block pointed by its grandchild (not its direct child). Thus, the optimizer should move A' from edge(A->B) to edge(B->Z) and edge(C->Y). The example in Figure 4.2-(c) can be optimized in two different ways. If A and B are considered as a whole unit, edge(B->Y) is the shortcut path,
and the graph can be optimized as shown in Figure 4.2-(d); otherwise, it can be optimized as shown in Figure 4.2-(e).

Detecting the existence of a shortcut path is not sufficient to determine that there is an outcome tolerant branch. The reason is that one of the blocks involved in the shortcut can modify a variable that is later used by instructions outside the block. That block needs to be replicated or the error could propagate outside the block. Next we show two examples:

(a) if \((*m > 0) \&\& (m < N)\) then X else Y
(b) if \((t=(*m > 0)) \&\& (m < N)\) then X else Y

In the example in (a), if \((*m>0)\) is mistakenly computed as True, but \((m<N)\) is False, we can safely ignore the error on \((*m>0)\) and take the Y path. However, if the error occurs to the example in (b), and \(t\) is used in Y, ignoring the error will result in a wrong value for \(t\) being propagated to Y, which may end up corrupting the system. To avoid this type of errors our compiler algorithm only considers blocks that are involved in a shortcut path and produce values that are only used by the block itself.
4.2 Compiler Algorithm

Our algorithm analyzes the control flow graph of the original program and extracts the shortcut paths and the related blocks. A shortcut graph always has a head node (block A in all the examples in Figure 4.2), one or more intermediate nodes (like B and C), two or more leaves (like X and Y), and one or more shortcut paths. Notice that in this paper we call a block to a single basic block or a list of basic blocks connected one by one with edges of unconditional branches.

Our algorithm has two phases: first a search of all potential shortcut graphs, and second, the optimization and appropriate placement of the replicas.

4.2.1 Shortcut Graphs Search

The searching process starts by classifying each block as an intermediate node or a leaf, and building an intermediate node set and a leaf set. A block is called “intermediate node” if it ends with a conditional branch and does not contain side effects (does not contain a function call, a memory write or generates a value used by another block). In addition, to avoid being trapped in loops, we require that none of the outgoing edges of an intermediate node is a loop backward edge. If the node does not classify as intermediate node, then it is considered a “leaf”, meaning that this block can be at the most an ending node in a shortcut graph.

At the same time we build the intermediate and leaf sets, we also build a separate head node set. A block is called ”head node” if it ends with a conditional branch and none of the outgoing edges is backwards, no matter it has side effects or not. Thus the head node set contains all intermediate nodes and some of the leaves.

After building the intermediate node set, the leaf set, and the head node set the shortcut graphs are built from bottom up by scanning the head node set repeatedly. We start by initializing an empty set “graph-head-set”, which will
contain temporary graph head nodes. For any node(A) in the head node set, we check its two children (see Figure 4.3):

1. If the two children are leaves, this node is added to the graph-head-set (Figure 4.3-(a)).

2. If one child is a leaf(X) and the other child is an intermediate node(B) and node(B) is already in the graph-head-set, node(B) is replaced by the current node(A) in the graph-head-set (Figure 4.3-(b)). We also check if the leaf(X) is a child or grandchild of node(B), in which case a shortcut path for node (A) is marked.

3. If the two children are both intermediate nodes((B) and (C)) and both are in the graph-head-set, nodes (B) and (C) are replaced by node(A) in the graph-head-set (Figure 4.3-(c)). We also check if (A) introduces new shortcut paths.

The scan continues until all the nodes in the head node set have been visited. Then, a node in the graph-head-set represents a graph led by this node together with the shortcut paths found. A final pass traverses the graph-head-set and removes those heads that do not contain any shortcut path.
4.2.2 Optimization

After the shortcut paths are found we start applying the optimization, but we first check when it is legal to perform it. In Figure 4.1-(b), our optimization will move the replica $A'$ from $\text{edge}(A\rightarrow B)$ to $\text{edge}(C\rightarrow Y)$. However, this is only legal if $A$ dominates $C$. Otherwise $A'$ may use undefined values in the new position. Thus to apply our optimization phase we first verify the domination relationship of all shortcut paths.

![Figure 4.4: Optimizing shortcut graphs.](image)

The goal of our optimization pass is to move replicas of the non-shortcut path down to the edge/s between the last child and the leaf/leaves. Next, we explain how this algorithm proceeds using the example in Figure 4.4. For each shortcut graph in the graph-head-set the algorithm finds all the shortcut paths (edge$(A\rightarrow X)$ in Figure 4.4-(a)), marks the replica (A’) on the other path as temporary (temp), and records the destination of the shortcut path (X). Next the optimization pass scans all the intermediate nodes in the shortcut graph in a top-down fashion, and moves temporary replicas from the incoming edges to all the outgoing ones, except to those where the recorded destination of the replica and the destination of the intermediate node that we are processing are the same (an example is shown in Figure 4.4-(b)). Notice that when an intermediate node has multiple incoming edges (as shown in Figure 4.4-(c)) we only move the replicas that appear on all the incoming edges. Also notice that this optimization pass
processes nodes top-down, and it does not treat multiple nodes as a single unit. Thus, for the example in Figure 4.2-(c), the optimized version after this pass will be the one shown in Figure 4.2-(e).

Finally note that A, B and C can contain computations like \((s+1) == 5\). In this case, if the computations are only used to determine the outcome of the branch, the computation replicas are also eliminated when the branch replica does not need to execute.

4.3 Evaluation

4.3.1 Performance

Figure 4.5 shows the performance speedup obtained when using boolean logic to eliminate replication and checks on outcome tolerant branches (Section 4). Three benchmarks (gzip, vpr, and perlbmk) achieve 7% performance gains, though the average speedup is 1.6% through all tested benchmarks. Notice that there is also a negative impact on vortex, where we observe more load/store instructions after the optimization, meaning that this optimization introduces additional register spills that hurt the benefit of less dynamic instructions. The reason for the additional register spills is that during the optimization we move some evaluations down to the control flow graphs, which may increase the life time of some registers.

4.3.2 Discussion on Reliability

We did not evaluate the change in fault coverage after this optimization, because we believe that the boolean logic optimization does not lose fault coverage. Taking the code structure in Figure 4.1-(c) for an instance, the discussion in Section 4.1 explains that if an error makes A flow a wrong branch, the error can be masked (if
Figure 4.5: Performance speedup with boolean logic optimization compared to baseline replication

the correct destination is $X$) or detected (if the correct destination is $Y$). A question rises, since $A$ does not have a shadow copy in some paths after the optimization, what would happen if an instruction opcode in $A$ is corrupted. Notice that our rule of constructing the short-cut path requires that $A$ must not contain update instructions changing values that are used by other blocks or outside, meaning that the only output of $A$ is the branch decision. Therefore, it makes little difference after optimization if the instruction opcode is corrupted: if the wrong opcode in $A$ causes updating a variable, the error will be caught when the variable is used and checked later because the shadow copy of this variable is not updated.

4.4 Conclusion

In this chapter, we propose the boolean logic optimization, which identifies some sections of code that are fault tolerant by themselves and removes some evaluations. The boolean logic optimization reduces the number of dynamic evaluations and checks but may increase register spills. If the benefit of reducing checks overcomes the loss from the extra register spills, performance gains can be seen. The experiment results show that this optimization improves performance by around 7% for three benchmarks, and less than 2% on average. Therefore, we need to use some heuristic to find which applications can benefit from this optimization.
and apply the optimization to only these applications. The heuristic can be based on how many patterns have been found, and how deep the short-cut graphs are. Obviously, the more patterns that the optimization can be applied to, the better performance we can gain; the shorter the short-cut graphs stand, the less register spills the optimization transformation will introduce. Overall, with a good heuristic, this optimization should always be applied, as it does not loose fault coverage, and can improve performance.
CHAPTER 5

Removal of Address Checks

This chapter presents the technique of removal of address checks. Since the technique transformation is very straightforward, we will focus on the motivation (Section 5.1) and evaluation (Section 5.2), and finally conclude this technique (Section 5.3).

5.1 Motivation

Recent experiments have shown that faults produce not only data corruption, but also events that are atypical of steady state operation and that can be used as a warning that something is wrong [23, 70]. Thus, we can reduce the overhead of the software approaches and trade reliability for performance by reducing the replication, hoping that the error will manifest with these atypical events.

In this chapter we consider the removal of address checks before load and store instructions. Errors in the registers containing memory addresses may manifest as segmentation faults. However, any fault-tolerant system must also include support for roll-back to a safe state and thus, on a segmentation fault we can roll-back and re-execute, and only communicate the error to the user if it appears again. However, by doing this the system will be vulnerable to errors, since some of these faulty addresses will access a legal space and the operating system will not be able to detect the error. Thus, this technique will decrease error coverage. Next, we discuss two techniques that the compiler can use to determine which
load and store instructions are most suitable for address check removal.

Address checks can be removed when there are later checks checking the same variable. For example, in Figure 5.1-(a), checking instructions (1-2) and (7-8) are checking the register \( r6 \). This makes the first check (1-2) unnecessary, because if an error occurs to \( r6 \) it will manifest as a segmentation fault or will be eventually detected by the checking instructions (7-8). We have observed many of these checks in the SPEC benchmarks due to the register indirect addressing mode, since the same register is used to access two fields of a structure, or because two array accesses share a common index. Removing these replicated checks can significantly reduce the software overhead.

Address checks can also be removed when the probability of error to the loaded value is small. This case appears in pointer chasing, where the data loaded from memory is used as the address for a subsequent load. An example is shown in Figure 5.2-(a) and (b). In this case, since the processor will issue the second load as soon as the first one completes, the probability of error is very small. In some cases, however, the value loaded by the first load is not exactly the one used by
the next load, if not that it may be first modified by an \texttt{add} instruction. This occurs when accessing an element of a structure that is different from the first one. In this case, the probability of error is higher, and the checking instructions will also determine if an error occurred during the computation of the addition. An example is shown in Figure 5.2-(c) and (d).

In next section, we evaluate the removal of the address checks for only the loads, or for both loads and stores. In the implementation, after removing these checks, we also remove instructions that were originally servering the checks but now are useless. We will show how much performance benefit we gain and how much fault coverage we lose.

\section*{5.2 Evaluation}

In this section we evaluate our proposed techniques. We first analyze our techniques statically (Section 5.2.1), evaluate performance (Section 5.2.2), and measure reliability (Section 5.2.3 and Section 5.2.4).

\subsection*{5.2.1 Static Analysis}

In this section we characterize load addresses depending on whether the register is checked by a later checking instruction (Covered), or if the register used by the
load was just loaded from memory (Loaded), as in the pointer chasing example of Section 5.1. All the remaining load addresses are classified as (Other). The breakdown is shown in Figure 5.3. On average more than 40% load addresses have nearby later checks on the same value. About 20% of the loads use registers whose contents were just loaded from memory. As we have discussed in Section 5, the probability of error of any of these addresses is very small, because the processor will likely issue the second load as soon as the first one completes. Also, if we assume a register safe platform these checks are unnecessary. For the remaining 40% of the addresses, an error in the most significant bits will be detected as a form of segmentation faults, but an error in the least significant ones can cause a silent error.

5.2.2 Performance

Figure 5.4 evaluates the performance benefit of our check removal technique: baseline Fully Replicated (FullRep), No checks for Address of Loads (NAL), No checks for Address of Load and Store (NALS). All numbers are normalized to (FullRep).

After we remove checks for address of loads (NAL), we get an average 20.2% speedup over the baseline Fully Replicated (FullRep). If we further remove checks for address of stores (NALS), we improve 4.6% more.
Figure 5.4: Performance of the different optimizations normalized against the original non-replicated code. (FullRep - Fully Replicated code, NAL - No Address checks for Loads, NALS - No Address checks for Load and Store)

Figure 5.5: Fault-detection rates break down. (O - Original non-replicated code, FR - Fully Replicated code, NAL - No Address checks for Load, NALS - No Address checks for Load and Store)
5.2.3 Fault Coverage

Since we remove all the checks for memory addresses, memory can be corrupted. In order to evaluate the loss of fault coverage, we use Pin [26] and inject faults to the binary file (excluding system libraries). We assume a Single Event Upset (SEU) fault model, that is, a single bit is flipped during the whole execution of the program. Although our detection mechanism will very likely detect multiple bit faults, the probability of multiple faults is much lower than SEU. Notice that to accurately model soft errors, one should use a HDL simulator and inject faults to buses, latches, combinational logic, and SRAM cells, among others. If this is done, many injected faults would be masked and a few would manifest as errors in the architectural status [58]. Here we report the result of injecting faults into the register and status flags. In effect, we are modeling only those errors that appear in the architectural status. We cannot inject faults that corrupt the program counter, so we cannot model that type of errors. However, notice we did not implement a mechanism to detect the illegal jumping in the control flow (as explained in Section 3.3) because the target of our optimization techniques is not the program counter. Finally, notice that a similar fault injection mechanism has been previously used by other software checking approaches [7, 53, 56, 68].

In our experiment a total of 300 faults are injected into four versions of each program: O - Original program, FR - Full Replicated version, NAL - No Address check for Load, NALS - No address check for Load and Store. Although both integer and floating point registers can be corrupted, in order to magnify the impact of the errors we only inject fault to the 8 32-bit integer registers and the status flags EFLAGS.

After injecting an error into the binary, the program is run to completion (unless it aborts) and its output is compared to a correct output. Depending on
the result the error will be categorized as:

- **(unACE)** the bit is unnecessary for Architectural Correct Execution [36];
- **(Detected)** the error is detected by our checking code;
- **(Self-Detected)**, the error is detected by the program assertions;
- **(Seg Fault)**, the error manifests as an exception or a segmentation fault;
- **(SDC)**, Silent Data Corruption, when the program finishes normally but the produced output is incorrect.

(SDC) is the first type of errors we want to prevent. Then, we also want to avoid (Self-Detected) errors and (Seg Fault) because it can not be immediately determined that the failure is due to soft error or some software bugs. But with proper support, if we can roll-back and re-execute, these faults can be recovered, so they are less harmful.

Figure 5.5 shows the experimental results for the random fault injection. Notice that the original program (O) has on average 75% (unACE) and less than 10% (SDC), which means that the software itself has a certain fault maskability.

After the program is replicated (FR), most (Seg Fault), (Self-Detected) and (SDC) go to the (Detected) category. (SDC) errors appear because some faults are injected before the value is used but after is checked. If we remove checks for addresses, reliability does not drop much. If we remove checks for load addresses (NAL), comparing to (FR), (SDC) increases from 0.36% to 1.08%, (Seg Fault) increases from 4.47% to 8.05%. If we also remove checks for store addresses (NALS), (SDC) rises to 1.44%, and (Seg Fault) rises to 9.02%.

Notice that after the program is replicated, about 20% (unACE) errors go to (Detected) category. These errors corrupt application states such that they
are detected by our inserted checks. However, these errors are masked by the application and do not affect final program outputs in the end. When we remove some checks, some of these errors return back to (unACE).

5.2.4 Fault Detection Latency

A software checking system always assumes a underlying checkpoint-rollback layer which is in charge of checkpoint and recovery when an error is detected. A checkpoint-rollback layer can hold a certain number of status updates after a checkpoint is taken. When an error is detected, the checkpoint-rollback layer will roll back to previous checkpoints and re-execute. It is very important that errors are detected before the number of status updates reaches the limit that this checkpoint-rollback layer can hold. Otherwise, the checkpoint snapshot may be corrupted, in which case the checkpoint-rollback layer will not be able to recover correctly even if the error is detected. In addition, if the number of status updates the checkpoint-recovery layer needs to keep is large, the performance penalty of a recovery will be high and will significantly impact the overall fault detection and recovery system. Thus, it is important to bound the fault detection latency.

After removing the address checks, the fault detection latency increases because errors are now detected by later checks, by the operating system, or by the application itself. In order to determine the detection latency, we inject faults to the programs and measure the distance between the instruction where the error was injected and the instruction where the error was detected (the error is detected by non-removed checks, segmentation faults, or assertion failures). Figure 5.6 shows the results. For each benchmark we show three bars. Each bar represents the errors that cause segmentation fault (Segfault), the errors that are detected by our inserted checks (Detected), and the errors that fail programmer
asserion (Self-detected), respectively. The total height of each bar shows the percentage of errors detected in each case with respect to the total number of injected errors. Each bar is broken down into different colors depending on the detection distance after the error is injected. We record a maximum of 100,000 instructions after an error is injected. If the error is not detected within 100,000 instructions, we let the program finish and report the final result of that execution.

Figure 5.6 shows that among (SegFault), on average 45.4% errors manifest as segmentation faults within 10 instructions, 78.8% within 100 instructions, 88.9% errors within 1,000 instructions, and 90.23% errors within 10,000 instructions. 7.9% of the segmentation fault errors (equivalent to 0.9% of the total injected errors) do not manifest within 100,000 instructions. Among (Detected), on average 73.9% errors are detected by non-removed checks within 100 instructions, 90.2% errors within 1,000 instructions, and 96.6% errors within 10,000 instructions. Only 3.03% of the errors that are detected by non-removed checks (equivalent to 1.0% of the total injected errors) do not manifest within 100,000 instructions. Among (Self-detected), 84.0% errors are not detected within 100,000 instructions. The reason is that most of these errors appear in gzip and bzip2 where assertions are checking if the compression or decompression correct, which takes long latency to find the error. In summary, the technique of removal address checks does not increase fault detection latency very much. After all address checks before loads and stores are removed, around 90% of the detected errors are detected within 1,000 instructions. And only 4.6% of the detected errors (equivalent to 2.1% of the total injected errors) are not detected within 100,000 instructions.
Figure 5.6: Fault-detection latency for the version where address checks before loads and stores are removed

5.3 Conclusion

In this chapter, we investigate a promising optimization strategy where we let the operating system to detect those cases that are likely to result in an abnormal behavior (segmentation fault in our study). This optimization reduces detection overheads since the error will be detected by the operating system with zero overhead. We found that errors in memory access addresses fall in this category. However, there is still the possibility that a faulty address is within a legal address range, in which case the error will not be detected by the operating system. Then we showed two cases where an error in memory access addresses is likely to be detected by later checkings.

In our evaluation, we blindly removed all address checks for loads (NAL), and then all address checks for both load and store (NALS). The results showed that for NAL, we gained performance over fully replicated program by 20.2% while
increasing SDC from 0.36% to 1.0%; for NALS, we gained 24.8% in performance and increased SDC to 1.44%. Overall, the loss in SDC detection coverage is acceptable, because the SDC of the original non-replicated program is as high as 6%. More importantly, considering that this SDC rate here is the probability of SDC over all manifested errors in architecture status, the real world SDC probability that we can observe will be the result of a multiplication of the SDC rate here by the raw SER rate and the probability that an error will propagate to architectural status. From Section 2.1.2 we see the raw SER rate is estimated to be 100 soft errors per $10^9$ hours of operation per chip in 2011. Want et al. [72] report that 15% of soft errors in processor state will propagate to architectural status. Since both multiplication factors are small numbers, our SDC rate of 1.44% is acceptable.

In addition, we measured the fault detection latency for errors that cause segmentation faults, assertion failure or are detected by our non-removed checks. A software checking system assumes an underneath checkpoint-rollback layer to do recovery. It is very important for the software checking system that errors will be detected shortly enough so that the underneath checkpoint-rollback layer is able to recover correctly. Our experiment results show that about 90% of the detected errors (the errors that finally cause segmentation faults, assertion failure or detected by our non-removed checks) can be detected within 1000 instructions. And only 4.6% of the detected errors (equivalent to 2.1% of the total injected errors) can not be detected within 100,000 instructions.
CHAPTER 6

ESoftCheck: Removing “Non-Vital” Checks

From Chapter 5, we know that some checks for addresses are not vital: removing those checks do not harm the fault coverage at all. One may have the following questions. How to identify those “non-vital” checks for addresses? Instead of removing all checks for addresses and losing certain fault coverage, can we remove only “non-vital” checks? Besides checks for addresses, are there other types of “non-vital” checks that can be removed? After such check removal, is the system still recoverable?

In this chapter, we explore the above questions and present ESoftCheck, a set of novel compiler optimizations that identify and remove the redundant “non-vital” checking instructions while keeping a level of reliability similar to the one of the non-optimized code. As a flexible software-based approach, ESoftCheck also provides knobs so that the user can trade reliability for performance. With ESoftCheck the user can i) define what are the commit points, and ii) specify the degree of trustiness of each operation. The distance between commit points is important, because the larger the number of instructions between commit points, the more redundant checks can be detected and removed. The degree of trustiness affects the removal of checks on variables that are data dependent through trusted operations.

This chapter is organized as follows. Section 6.1 presents an overview of ESoftCheck, Section 6.2 discusses in detail the compiler algorithms that ESoftCheck uses, Section 6.3 discusses how ESoftCheck optimizations compares with previous
6.1 Overview of ESoftCheck

In this section we present an overview of ESoftCheck. Section 6.1.1 discusses the type of checks that ESoftCheck can optimize, Section 6.1.2 describes the knobs provided by ESoftCheck, and Section 6.1.3 discusses some of the issues that appear.

6.1.1 Type of Redundant Checks

The added instructions in the augmented code of Figure 3.1-(b) can be classified as either shadow copies of the original instructions (instructions 1, 4 and 5) or error checking instructions (instructions 2, 3 and 6-9). The key idea of ESoftCheck is the detection of redundant error checking instructions that are non-vital and that can be removed while maintaining the same or similar fault coverage. ESoftCheck uses compiler techniques to optimize four types of checks. Next, we describe them (a more formal description is presented in Section 6.2):

1. **Checks covered by a later check to the same register.** A check of a register is redundant if it is always followed (covered) by other checks of the same value in the register at the time of the check, and the register has not been modified in between the two checks. In such a case, one can remove the first check, since an error will be discovered by the subsequent check(s). The example in Figure 6.1-(c) shows the optimized code of Figure 6.1-(b) (the same as Figure 3.1-(b)). Instructions 8 and 9 check the same register as instructions 2 and 3, and the value of r6 does not change between the two checks. Therefore, instructions 2 and 3 can be removed.
Notice that by removing the instructions 2 and 3 in Figure 6.1-(c), it is possible to load from a wrong address (what will result in a wrong value in both \texttt{r3} and \texttt{r3'}) or cause a segmentation fault. The error in the load address will be detected when checking \texttt{r6} in instruction 8 and 9, and we will know any registers that have consumed the loaded value may be wrong. Section 6.1.3 explains how to handle segmentation faults resulting from transient errors.

2. **Checks covered by a later check of a different register.** A check of a register is redundant when it is followed (covered) by a check of a different register whose value is a function of the first register.

An example is shown in Figure 6.2. Figure 6.2-(a) shows the original code, and Figure 6.2-(b) shows the replicated code. In this example, it is possible to remove instructions 1 and 2 that check register \texttt{r1}, because \texttt{r4} is computed by adding a constant to \texttt{r1}. An error in \texttt{r1} propagates to \texttt{r4}, and is detected when \texttt{r4} is checked.
3. Checks of loop induction variables and loop invariants. Checks inside loops can be made redundant and removed by adding covering checks at the loop exits, thus reducing the dynamic check count. Figure 6.3-(a) shows a loop where register r1 contains an induction variable. The corresponding replicated code is shown in Figure 6.3-(b) where checks at the taken and fall through paths of the conditional branch (instructions 4, 5, 6, and 7) verify that the loop executes the correct number of iterations.

Figure 6.3-(c) shows the ESoftCheck code, where instructions 1 and 2 that check register r1 have been moved outside the loop because the compiler has determined that r1 contains an induction variable, and any error in the loop will also propagate outside, where it will be detected.

4. Checks already protected by the hardware. The register file of current platforms such as Intel Itanium [30], Sun UltraSPARC [21] and IBM Power4-6 [5] are already hardware-protected by parity or ECC or can be protected with cost-effective hardware mechanism [8, 15, 19, 33]. In these platforms, that we call register safe platforms, errors altering the values in registers are detected by the hardware. Notice that we assume that the path to and
from the register file is also hardware protected. Otherwise, there will be a window of vulnerability.

Two types of checks are redundant in this case:

- **Type I**: Checks of registers defined by loads that have not been modified by arithmetic or logic operations can be removed. In addition, the registers defined by loads do not need to be replicated, saving a copy instruction and reducing register pressure. An example is shown in Figure 6.4. Figure 6.4-(a) shows the original code, a pointer chasing code, where the data loaded from memory is used as the address of the next load. Figure 6.4-(b) shows the replicated code and Figure 6.4-(c) shows the optimized ESoftCheck code. ESoftCheck removes instructions 3, 4 and 5, because errors in register $r_1$ are detected by the hardware.

- **Type II**: Checks followed by a check to the same register that has not been modified. In this case, we delete the second check, rather than the first one: if the first check succeeds, then the register is correct
<table>
<thead>
<tr>
<th>Original code</th>
<th>Replicated code</th>
<th>ESoftCheck</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ld r1=[r2]</code></td>
<td><code>ld r1=[r2]</code></td>
<td></td>
</tr>
<tr>
<td><code>cmp r2, r2</code></td>
<td><code>cmp r2, r2'</code></td>
<td></td>
</tr>
<tr>
<td><code>jne faultDet</code></td>
<td><code>jne faultDet</code></td>
<td></td>
</tr>
<tr>
<td><code>ld r3=[r1]</code></td>
<td><code>ld r3=[r1]</code></td>
<td></td>
</tr>
<tr>
<td><code>mov r1'=r1</code></td>
<td><code>mov r1'=r1</code></td>
<td></td>
</tr>
<tr>
<td><code>cmp r1, r1'</code></td>
<td><code>cmp r1, r1'</code></td>
<td></td>
</tr>
<tr>
<td><code>jne faultDet</code></td>
<td><code>jne faultDet</code></td>
<td></td>
</tr>
<tr>
<td><code>mov r3'=r3</code></td>
<td><code>mov r3'=r3</code></td>
<td></td>
</tr>
</tbody>
</table>

Figure 6.4: Check removal when register file is safe.

at the place where the second check occurs. An example is shown in Figure 6.1-(d) where the later check of r6 is removed and r6’ can be deallocated earlier, reducing register pressure.

Notice that to simplify the discussion our examples only contain two checks and simple data dependences, but ESoftCheck can detect more than one redundant check linked by a chain of data dependences.

### 6.1.2 Knobs

ESoftCheck provides two types of knobs so that the user can trade reliability for performance. The user can trade the number of redundant checks that can be removed based on the frequency of checkpoints and the trustiness of operations.

#### Checkpoints

A fault tolerant system requires a checkpoint mechanism that saves snapshots of the application state where to roll back if an error is detected. The *commit points* are the instructions in the application where a new checkpoint may be taken so that the space used by the previous checkpoint can be released. To be able to properly recover, a checkpoint must not contain corrupted data. Thus, at commit point we need to make sure that any possible error has been detected.
The redundant checks that ESoftCheck can remove depend on the location of the commit points. For example, given two checks to the same register the first check cannot be removed if there is a checkpoint between the two checks. An example illustrating this situation is shown in Figure 6.5-(a) and (b), where the check operator consists of two instructions: a “comparison” instruction to compare the contents of the register in the original code with the contents of its replica and a “conditional branch” to an error handler if a mismatch is detected. Similarly, when a commit point is inside a loop, checks of loop induction variables and of loop invariants cannot be moved outside the loop. The only checks that are not affected by commit points are those removed because the register file is safe. An example is shown in Figure 6.5-(c) and (d). The reason is that in this case ESoftCheck keeps the first check and deletes the last one. After the first check it is the hardware responsibility to detect any error to the register.

![Figure 6.5: ESoftCheck in the presence of checkpoints.](image)

ESoftCheck provides knobs so that the user can specify the location of the commit points. Notice that in general the more instructions between checkpoints, the more likely it will be for ESoftCheck to find redundant checks that can be removed, resulting in a larger reduction of the overheads. For this paper we evaluate two different checkpoint frequencies, described in Section 6.2.4.
Degree of trustiness

As explained in Section 6.1.1, a check can be removed when it operates on a register whose value is a function of an earlier checked register. However, this approach can mask some errors. For instance, if we have \( \text{mul } r4=r1,r3 \) and \( r3 \) is zero, by only checking \( r4 \) we will not detect if there is an error in \( r1 \). Thus, we define *trusted* operators as those that have a low chance of masking errors. ESofCheck can provide knobs so that the user can specify which operators are to be considered *trusted* so that variable \( a \) can be checked through variable \( b \), when variable \( b \) depends on \( a \) through a chain of dependences that only involves *trusted* operators. In general, arithmetic and shift operators are considered trusted. For logic operators the probability of error propagation will depend on the number of 0’s and 1’s.

An interesting situation appears with the conditional move operator: \( \text{cmov } r4, r3, \text{cond} \) copies register \( r3 \) to \( r4 \) if \( \text{cond} \) is true; otherwise it does not do anything. We consider that it is not safe to check register \( r3 \) by checking the contents of \( r4 \), because when \( \text{cmov} \) does not perform the copy, an error in \( r3 \) will not be detected by checking \( r4 \). On the other side, since \( \text{cond} \) is computed as the result of a comparison instruction that executes before the \( \text{cmov} \), it would be possible to check the operand registers of the comparison through a check of \( r4 \). However, since the comparison only has two possible outcomes, an error in the comparison operand register has a high probability of being masked. Thus, we consider the \( \text{cmov} \) operator not trusted.

6.1.3 Issues

ESofCheck can increase the number of segmentation faults with respect to the Baseline Fully Replicated codes. The reason is that by removing redundant checks
it is possible that some errors will manifest as segmentation faults before the error is detected, that is, before the check that has not been removed is executed. In these circumstances we will not know if the segmentation fault is the result of a programming error or of a soft error. However, since the operating system knows where the segmentation fault occurred and a fault tolerant system must have a mechanism to roll-back to a safe state, we can roll-back and re-execute. If the same error appears the operating system will notify of an error to the user; however, if the error does not appear again, we can consider it was due to a soft error or a software bug. Notice that the number of segmentation faults will not increase in the case of optimizations due to register safe platforms, as the check deleted is the last one.

6.2 ESoftCheck Framework

Our optimizations are implemented as passes on the LLVM intermediate level [22], which is a SSA representation [14]. In this section we present the optimization algorithms for the checks covered by the same or different registers (Section 6.2.1), checks before loop induction variables and loop invariants (Section 6.2.2), checks already protected by the hardware (Section 6.2.3).

6.2.1 Covered Checks

A check $c_1$ of register $r_1$ is covered by another check $c_2$ of register $r_2$ when

I $c_2$ postdominates $c_1$.

II Either $r_2=r_1$ or $r_2$ depends on $r_1$ through a chain of data dependences that only involve trusted operators.

III There is no update to register $r_1$ on any path between the two checks.
Trusted operator makes sure that any error that occurs to its operand will result in an error to its result. Strictly speaking, trusted operator must be a one-to-one mapping from its operand to its result. Otherwise, there is a chance that an error in its operand does not incur an error to its output, thus the error will not be detected as a result. However, if extreme fault coverage is not required, one may want to relax the requirement to be able to remove more checks. As we have discussed in Section 6.1.2 ESoftCheck provides users knobs to determine which operators are trusted, such that users can trade off between reliability and performance.

If \( r_1 = r_2 \) we say \( c_1 \) is directly covered by \( c_2 \); otherwise we say it is indirectly covered. A covered check \( c_1 \) can be eliminated if there is no commit point on any path between \( c_1 \) and \( c_2 \). An example is shown in Figure 6.6. On Figure 6.6-(a) and (b) \( c_1 \) is covered by \( c_2 \), but on (b) \( c_1 \) cannot be removed because of the commit instruction. On Figure 6.6-(c), \( c_1 \) is not covered because \( c_2 \) does not postdominate. On Figure 6.6-(d), \( c_1 \) is covered because the combination of \( c_2 \) and \( c_3 \) postdominates \( c_1 \).

\[
\begin{align*}
&\text{(c1)check } r \\
&\text{(c2)check } r \\
&\text{t }= r \text{ op } 8 \\
&\text{(c2)check } t
\end{align*}
\]

Figure 6.6: Examples of covered and non-covered checks

Next, we present the algorithm to detect covered checks that can be removed (Section 6.2.1) and then discuss how to apply it based on the support for checkpoint and rollback (Section 6.2.4).
Algorithm To Remove Covered Checks

The algorithm to detect and remove Covered checks is a combination of the backward data-flow problem (that determines the postdomination property) and a data-dependence problem (that records data dependences through trusted operators). It is similar but different from classic Common Sub-expression Elimination (CSE) [34], in the following aspects:

I We can optimize the data-dependence case where two checks are checking different registers, while CSE can not optimize if two expressions have different operands.

II We need to preserve the latest check for an error, while CSE preserves the earliest evaluation of an expression.

III Commit point will kill all available checks in our case, while CSE does not have such a powerful killer.

We define that \( \text{check}(r) \) is available at point P if on every path from the program end to P, there is a \( \text{check}(t) \) at point P’ (t depends on r through a chain of trusted operators, or \( t = r \)), and there is no update to r and no commit point in between. To determine if a \( \text{check}(r) \) can be removed ESoftCheck will determine if \( \text{check}(r) \) is available right after it appears in program order using the algorithm described next.

The flow analysis of available checks will be discussed in two parts. First, we present the intra-basic block (local) analysis, and then the global flow analysis.

• Local flow analysis.

Let \( AC_{AF}(I) \) and \( AC_{BF}(I) \) be the set of Available Checks right after and right before instruction I (in program order), respectively. Let \( Chk_{Ins}() \) be
the instruction transfer function which computes $AC_{BF}$ in terms of $I$ and $AC_{AF}$:

$$AC_{BF}(I) = Chk\_Ins(I, AC_{AF}(I)).$$

$Chk\_Ins()$ is defined as follows:

- If $I$ is $check(r)$, $AC_{BF}(I) = AC_{AF}(I) \cup \{check(r)\}$

- If $I$ is $r=OP(t)$ and OP is a trusted operator, we will make $check(t)$ available if $check(r)$ is already available right after $I$. The reason is that an error in $t$ will be detected by $check(r)$. In addition, since $r$ is updated we kill the availability of $check(r)$. With this we propagate available checks through chains of data dependence.

$$AC_{BF}(I) = \begin{cases} 
(AC_{AF}(I) \cup check(t)) - \{check(r)\}, & \text{if } check(r) \in AC_{AF}(I) \\
AC_{AF}(I) - check(r), & \text{otherwise}
\end{cases}$$

- If $I$ is $r=OP(t)$ and OP is not a trusted operator, we kill the availability of $check(r)$. $AC_{BF}(I) = AC_{AF}(I) - \{check(r)\}$

- If $I$ is commit point, all the available checks are killed, as they cannot propagate across commit commits. Thus, $AC_{BF}(I) = \emptyset$

- Otherwise, $AC_{BF}(I) = AC_{AF}(I)$

- **Global flow analysis.**

  Let $AC_{IN}(B)$ and $AC_{OUT}(B)$ be the set of available checks on entry to and exit of basic block $B$, respectively. Let $Chk\_Blk()$ be the basic block transfer:

  $$AC_{IN}(B) = Chk\_Blk(B, AC_{OUT}(B)).$$

  Assuming that the basic block $B$ contains the instruction sequence $I_1$, $I_2$, ... $I_n$, we define $Chk\_Blk()$ as
\[ AC_{IN}(B) = AC_{BF}(I_1) \]
\[ = Chk_{Ins}(I_1, Chk_{Ins}(I_2, \ldots Chk_{Ins}(I_n, AC_{OUT}(B)) \ldots)) \]

Figure 6.7 shows an example. For instance, at instruction 3 \((r_3=r_1+16)\), check \((r_3)\) is killed. In addition, check \((r_1)\) is made available because \(r_3\) depends on \(r_1\) through a trusted operator and check \((r_3)\) is available right after instruction 3. Thus, when instruction 1 is processed, we find that check \((r_1)\) is available right after it. As a result check \((r_1)\) at instruction 1 can be removed.

To guarantee the postdomination property a check is available at the exit of a basic block only if the check is available on the entries of all the successor basic blocks. Then, the data flow equations are:

\[ (a) \quad AC_{OUT}(B) = \bigcap AC_{IN}(S), \quad \text{over all successors } S \text{ of } B \text{ in the data flow graph} \]
\[ \text{and} \]
\[ (b) \quad AC_{IN}(S) = Chk_{Blk}(S, AC_{OUT}(S)) \]

59
We use an iterative approach to solve the data-flow equations (a) and (b) [34]. Notice that the check removal can be applied while the available check analysis is being applied. Initially, we define $AC_{OUT}$ for all the program exit blocks $(ExitB)$

$$AC_{OUT}(ExitB) = \emptyset$$

and for the non-exit blocks $(NonExitB)$ as $AC_{OUT}(NonExitB) = \bigcup AC$ where $\bigcup AC$ is the union of all possible available checks. The data-flow analysis iterates until there is no change in any $AC_{OUT}(B)$.

When it finishes, we have $AC_{OUT}(B)$ for every basic block $B$ and $AC_{AF}(I)$ for every instruction $I$. Thus, given a check($r$), if check($r$) $\in AC_{AF}(I)$, this check is covered and can be safely removed. Finally, notice that the check removal can be applied while the available check analysis is being applied.

**An efficient and practical way to compute the block transfer function.**

From Equation 6.2, we can see that the transfer function function $Chk\_Blk()$ will be called every time when the available checks for block $S$ $AC_{OUT}(S)$ are updated. Thus it is very inefficient to follow Equation 6.1 and sweep every instruction in the block when $Chk\_Blk()$ is called. We will present an efficient way that calculates $Chk\_Blk()$ for each block only once.

First we define $Gen(i)$ and $Kill(i)$ for each instruction $i$. A check($r$) generates an entry in $Gen(i)$. A definition of a variable $r$ generates an entry in $Kill(i)$. A “commit point” kills all the checks. To remember the data dependences through trusted operators, we introduce a third set that we call $Cover(i)$. If $t$ is defined based on $r$ through a trusted operator ($t = OP (r)$), then a check($r$) will be available if check($t$) is available. Thus, when we find such type of instruction we add a pair ($r/t$) to $Cover(i)$, meaning that check($r$) is indirectly covered by check($t$).

Figure 6.8 shows an example of how to define the three sets. For instance, instructions 5 (check ($r_2$)) generates a check of $r_3$, whereas instruction 4 ($r_3 = r_1 + 82$), kills $r_3$ and adds the pair ($r_1/r_3$) to $Cover(i)$ meaning that check($r_1$)
is indirectly covered by \texttt{check(r3)}. This information is necessary because when processing instruction 4 to compute the available checks, \texttt{check(r1)} will be made available if \texttt{check(r3)} is available. By making \texttt{check(r1)} available and earlier \texttt{check(r1)} (in program order) will be covered and could be removed.

Next we compute \textit{Gen(B)}, \textit{Kill(B)}, and \textit{Cover(B)} for each basic block \textit{B}. For that we traverse backwards all the instructions in block \textit{B}, and recursively apply the transfer functions in Figure 6.9. An example of applying the transfer function is shown on the right part of Figure 6.8. The scan starts at instruction 7 and proceeds backwards. Notice that when instruction 2 is processed we know for sure that \texttt{check(r1)} is available, so we can remove the pair \texttt{(r1/r3)} from the \textit{Cover} set and add \texttt{r1} to \textit{Gen} set. When instruction 1 is processed, we see that \texttt{check(r1)} that covers \texttt{check(r1\_inc)} is available, so we can add an entry \texttt{check(r1\_inc)} to \textit{Gen}.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig6.8.png}
\caption{An example of computing block transfer function \textit{Chk\_Blk()}.}
\end{figure}

Now we can define \textit{Chk\_Blk()} though as \textit{Gen()}, \textit{Kill()}, and \textit{Cover()}. 

61
Check(r1)

<table>
<thead>
<tr>
<th>Type of i2</th>
<th>Transfer function for entire block</th>
</tr>
</thead>
<tbody>
<tr>
<td>Check(r1)</td>
<td>( Gen = Gen(i1) \cup { r1 } ) ( \text{Kill} = \text{Kill}(i1) - { r1 } )</td>
</tr>
</tbody>
</table>
| Gen(i2)=\{r1\} Kill(i2)=\{} Cover(i2)=\{} | Gen = (Gen(i1) − \{ r1 \}) \cup \{ d_i | \text{if } r1 \in \text{Gen}(i1) \} | \( \text{Kill} = \text{Kill}(i1) + \{ r1 \} \)
| r1=OP(di) OP is safe operator | \( \text{Cover} = \text{Cover}(i1) - \{ (r1/v) | (r1/v) \in \text{Cover}(i1) \} \) \cup \{ (d_i/v) | (r1/v) \in \text{Cover}(i1) \} \cup \{ d_i/r1 | (r1/v) \notin \text{Cover}(i1), r1 \notin \text{Gen}(i1) \} |
| Gen(i2)=\{} Kill(i2)=\{r1\} Cover(i2)=\{d/r1\} | \( \text{Gen} = \phi \) \( \text{Kill} = \{ \text{ALL} \} \) | \( \text{Cover} = \phi \) |

*Instruction i2 is right before instruction i1 in program order.

Figure 6.9: Rules to compute \( \text{Gen}(B) \), \( \text{Kill}(B) \), and \( \text{Cover}(B) \).

\[
AC_{IN}(B) = (r | r/d) \in \text{Pend}(B), d \in AC_{OUT}(B))
\]

\[
\bigcup Gen(B)
\]

\[
\bigcup (AC_{OUT}(B) - \text{Kill}(B))
\] (6.3)

With the new Equation 6.3, the old data flow Equation 6.2 shall be updated as follows:
(a) \( AC_{OUT}(B) = \bigcap AC_{IN}(S) \), over all successors \( S \) of \( B \) in the data flow graph and

(b) \( AC_{IN}(S) = (r | (r/d) \in Pend(S), d \in AC_{OUT}(S)) \)

\[ \bigcup Gen(S) \]

\[ \bigcup (AC_{OUT}(S) - Kill(S)) \] (6.4)

Since \( Gen(S), Kill(S) \) and \( Pend(S) \) are fixed after scanning of all instructions in block \( S \) for one time, Whenever \( AC_{OUT}(S) \) is updated and we need to re-calculate \( AC_{IN}(S) \), we can directly use these three sets without scanning every instruction of the block again.

### 6.2.2 Loop Checks

Our algorithm detects loop induction variables and loop invariants and move the checks of these variables outside the loop. Induction variables are variables whose successive values form an arithmetic progression in a loop. In the SSA form, loop induction variables are defined by cycles involving \( \phi \) (Phi)-functions [74].

In our implementation we first apply loop strength reduction [34] so that some operations like multiplications are replaced by additions. Then, we scan \( \phi \) (Phi)-functions in the loop header node. Given \( r = \phi((pre\_header, r0), (backedge, r2)) \), if \( r2 \) is defined as \( r \) plus (or minus) a loop constant, we consider \( r \) as a loop induction variable. Then, the check of the induction variable inside the loop can be moved outside the loop. Similarly, checks of loop invariants are also moved outside. This optimization is only applied to loops that do not contain commit points.
6.2.3 Hardware Protected Checks

As explained in Section 6.1.1-4, on register safe platforms there are two types of checks that can be optimized.

- **Case 1.** If register $r$ is defined by a load, $r$ does not need to be replicated, and checks on $r$ can be removed.

- **Case 2.** After a check $c$ on register $r$, $r'$ can be deallocated, and any later check on register $r$ can be removed.

We unify the two cases by defining that **register $r$ is safe at point $P$** if $r$ is defined by a load or there is a check on $r$ at point $Q$, such that $Q$ dominates $P$ and $r$ does not change on any path between point $Q$ and point $P$. If we know register $r$ is safe at point $P$, a check on $r$ (at $P$) can be removed, and the use of $r'$ (at $P$) can be replaced by $r$.

Notice that safe register optimization is independent of where commit points are (See Section 6.1.2).

The problem of determining what registers are safe at a given point is a forward data-flow problem. Similar to the available expression problem, we use $\text{GEN}_{SR}(B)$ to denote the set of generated safe registers in block $B$ that are not redefined at its exit and $\text{KILL}_{SB}(B)$ to denote the set of registers that are killed by block $B$. To compute $\text{GEN}_{SR}(B)$, we scan block $B$ from beginning to end, accumulating the generated safe registers (the register is defined by a load or there is a check on it), and deleting those that are assigned new values in the block. Then, we use $\text{SR}_{IN}(B)$ and $\text{SR}_{OUT}(B)$ to represent the set of safe registers on entry to and exit from block $B$, respectively. A register is safe on entry to block $B$ if it is safe at the exits of all predecessor blocks. A register is safe at the exit from a block if it either generated in the block and not subsequently killed in it,
or if it safe on entry to the block and not killed in it. The system of data-flow equations is

\[ SR_{IN}(B) = \bigcap SR_{OUT}(S_i), \text{ where } S_i \text{ is predecessor of } B. \]

\[ SR_{OUT}(B) = GEN_{SR}(B) \cup (SR_{IN}(B) - KILL_{SR}(B)) \]

For the data-flow analysis, we initialize \( SR_{IN}(Entry) = \emptyset \), and for non-entry block B \( SR_{IN}(B) = U_{SR} \) where \( U_{SR} \) is union of all generated safe registers.

### 6.2.4 Knobs

- **Checkpoints**: ESoftCheck determines the covered checks that can be eliminated based on the location of the commit points. As explained in Section 6.1.2 checks cannot be delayed across commit points. In this paper, we evaluate two checkpoint frequencies. To compare with previous proposals, we follow their approach: a program is considered correct if its output is correct (assuming memory-mapped I/O), that is, if all the stores have executed correctly [47, 53–55]. Under this approach stores, function calls and function returns are commit points and checks cannot be delayed across commit points. As a result, checks can only be removed before loads. We call this model \textit{MemUnPolluted}.

To show the flexibility of ESoftCheck we also evaluate another system that we call \textit{MemCheckpoint}, where commit points are function calls and function returns. With MemCheckpoint stores are not commit points and memory can be corrupted with wrong values. Thus, a mechanism for memory checkpointing either in software [6] or in hardware, such as ReVive [48] or SafetyNet [63] is necessary.

- **Degree of Trustiness**: With ESoftCheck the user can specify which are the trusted operators. For the experiments in Section 6.4 arithmetic, shift
and logic operators are trusted operators, while the conditional move operators are not. Due to space limitations, we do not evaluate the impact on performance or reliability of the degree of trustiness.

6.3 Compared to Previous Work

6.3.1 Difference From DBCE

Instruction-level replication can also be implemented in hardware. From Section 2.2.1, we see that the Redundant Multi-Threading schemes run two copies of the same thread and check with each other. Depending on detection or recovery, different schemes do checks at various places.

In AR-SMT [57] two threads are checked before an instruction commits at the trailing thread side. The committed state of the trailing thread can be used as a checkpoint for recovery. SRT [52] and CRT [35] checks only committed stores and uncached loads, assuming that an error will eventually propagate to stores or uncached loads or it will not affect the program result at all. SRT and CRT does not support recovery, in that every architectural state is committed before check. When an error is detected, no one knows which architectural state is correct. When error recovery is considered, the number of checks increases significantly because the trailing thread is used to recover. Thus, every instruction in the trailing thread needs to be checked before it commits. To reduce the number of checks, SRTR [67] proposes the use of “Dependence-based Checking Elimination”, that exploits register dependence chains, so that only the value of the last instruction in a chain is checked. CRTR [16] finds that instructions that mask operand bits may mask faults and limit the use of dependent chains. It proposes Death- and Dependence-Based Checking Elision (DDBCE), which chains
Figure 6.10: Example of difference between ESoftCheck and CRTR on Dependence-based checking elision.

a masking instruction only if the source operand of the instruction dies after the instruction.

Our optimization to remove the covered checks by a later check to a different register (second type of checks in Section 6.1.1) exploits a similar idea as DECE (DDBCE is not applicable). The reason is that the checks ESoftCheck tries to remove are before loads, stores or other synchronization instructions, so more considerations need to be taken into account when optimizing these checks.

Taking Figure 6.10-(a) as an example, in CRTR, if r6’s consumer r4 masks r6, but r6 only has one consumer, check(r6) can be removed. Because even if there is an error in r6 and the error is not caught by check(r4), the error is not visible beyond this point and has not bad impact at all. But in ESoftCheck (example shown in Figure 6.10-(b)), check(r6) must be followed by a synchronization instruction that uses r6 (ld r1=[r6] in this example). If check(r6) is removed, in case an error happens to r6 that makes “ld r1=[r6]” fetch a wrong value, and check(r4) does not catch this error, then the error that propagated to r1 will not be detected and may cause trouble.
6.3.2 Benefit for Software-Only and Hybrid Solutions

ESoftCheck compiler techniques are beneficial for both software-only and hybrid (software/hardware) fault tolerant solutions.

Software-only solutions applied to single-core [7, 53] benefit from a lower register pressure and a reduction in the number of comparison and branch instructions executed. Since ESoftCheck maintains full fault coverage, it can be applied in combination with some partial replication schemes. For example, PROFit [56] and Spot [55] selectively replicate some sections of a program. ESoftCheck optimizations can be applied to the replication on those selected sections to achieve better performance.

For multi-core environment [68], checks are inserted before system calls for I/O operations and shared memory access operations. The leading thread sends the value that needs to be checked to the trailing thread, and the check is performed at the trailing side. In most cases, the leading thread continues without waiting for the checking acknowledgement. But in some cases, if the error may affects I/O or other threads, the leading thread has to wait for the checking acknowledgement before proceeds. Once the checking fails, the leading thread fail-stops immediately. ESoftCheck can not remove these fail-stop checks, because otherwise the error may already propagate to I/O or outside before it is detected. In SRMT’s definition, fail-stop checks check volatile loads/stores and shared stores. ESoftCheck can optimize non-failstop checks. Figure 6.11-(a) gives an example of how covered-check removal optimizes on SRMT. The check on r1 can be removed because an error in r1 will be detected by the check on r3. Therefore, one data communication and one check are saved. [68] reports that the leading thread dominates the SRMT execution time, so the save on trailing checks may not help the performance much. But the save on cross-core communication bandwidth is projected to be
Figure 6.11: Example of applying covered-check removal for SRMT.

important for multi-core processors. As for the example shown on Figure 6.11-(b), since [r1] is a memory-mapped I/O address, a fail-stop check on r1 must be inserted before leading thread performs the load. In this case, ESoftCheck can no optimize.

In the case of hybrid solutions [47, 54], ESoftCheck reduces the number of hardware checks, and lowers register pressure.

### 6.4 Evaluation

In this section we evaluate our proposed techniques. We first characterize the number of static checks that can be removed (Section 6.4.1), evaluate performance (Section 6.4.2), and measure reliability (Section 6.4.3).

#### 6.4.1 Characterization of Static Checks

In this section, we characterize the static checks that can be removed. A breakdown is shown in Figure 6.12. There are four bars for each application. The first
three bars characterize the checks based on the type of instruction they guard: load (ld), store (st), and function call and return (other). The last bar (to) corresponds to the sum of all the checks in the three previous bars. The bars are normalized to the total number of checks for each type of instruction. A check is categorized according to the reason why it can be removed: (i) because it is covered by another check to the same or different register (Covered), (ii) because it is before a loop induction variable or loop invariant (Loop), (iii) because the register file is safe (SafeReg). The checks that cannot be removed appear as (NotRem).

For the characterization we assume the MemUnPolluted model described in Section 6.2.4.

Notice that a given check may belong to Covered and RegSafe at the same time. However, in our characterization this check appears as Covered, since the pass that detect Covered checks is applied first. The first observation we make is: (1) with the MemUnPolluted model only the covered checks before loads can be removed. The removed checks in this case account for 42.8% of the checks before loads and 19.1% of the total number of checks; (2) the fraction of checks that guard loop induction variables and loop invariants is very small; (3) when the register file is safe, an average of 32.1% of the checks can be removed.
6.4.2 Performance

Figure 6.13 shows the performance benefit of different optimizations: Fully Replicated (FullRep), removal of checks that are covered by another check to the same or different register (Covered), removal of checks because the register file is safe (SafeReg), and removal of checks before loop induction variables and invariants (Loop). The first five bars of each application correspond to the MemUnPolluted model where stores, function calls and function returns are considered commit points. In addition, we also show performance numbers for the MemCheckPoint (MemChkpt) model, where commit points are function calls and function returns.

The Fully Replicated code (FullRep) is on the average 2.16 times slower than the original code. This large overhead is due to several reasons:

I Register pressure. the replicated code needs twice more registers than the original application, and the x86 ISA only has 8 registers available to the compiler.

II The additional instructions.

Previous works have published smaller overheads for FullRep [7,53,54] but in that work the target machines were Itanium or PowerPC platforms that have a larger number of registers. On the average, (Covered) runs 6.9% faster than (FullRep), but only checks before loads can be removed in this case. We have also removed all the checks before loads (not shown in the Figure), and found out that the average normalized execution time is 1.82 (versus 2.02 of Covered). Thus, (Covered) reduces 41.2% of the overhead introduced by checking the load address.

The safe register optimization (SafeReg) obtains a high performance benefit because it can remove redundant checks and reduce register pressure. On the average, (SafeReg) runs 24.6% faster than (FullRep).
Figure 6.13: Performance of the different optimizations normalized against the original non-replicated code.
The optimization before induction variables and loop invariants (Loop) results in little performance gain. The reason is that we impose many constraints and so the optimization is applied to only a few loops.

When we combine the three optimizations together (Covered+Loop+RegSafe), there are some checks that can be optimized with both (Covered) and (SafeReg). However, since (SafeReg) removes the second check, reducing register pressure, when both (Covered) and (SafeReg) can be applied, we apply the (SafeReg) optimization. The combination of the three optimizations achieves on average 27.1% speedup compared to (FullRep), resulting in a code that runs 1.70 times slower than the original code. Under the MemCheckPoint model (last two bars in Figure 6.13) where checks before store can also be removed, the (Covered) optimization has on average 10.7% speedup over (FullRep) and the combination of all optimizations achieves on average 31.7% speedup compared to (FullRep).

Notice that FullRep corresponds to state of the art approaches such as SWIFT [53] that use only software checking and no special hardware for error detection. Under the MemUnPolluted model, when applying all our optimizations (Covered+Loop+SafeReg) the overhead is reduced from 116% to 70%, and the only hardware support that we need is parity or ECC in the register file (that is already in many processors).

Notice that with 70% overhead, we have full protection on integer, floating point instructions, as well as conditional branches. And the memory is guaranteed not to be corrupted. The only hardware support we need is parity or ECC in register files, which is already there in most server processors in today’s market. If we have advanced hardware support, such as store value queue [47, 54] and branch protector [47], we can achieve better performance in that the hardware takes over the job of checking for store and branches.
6.4.3 Reliability

To evaluate the reliability of our optimizations, we use the same fault injection method described in Section 5.2.3: we use Pin [26] and inject faults to the binary file (excluding system libraries) assuming a Single Event Upset (SEU) fault model.

In our experiments a total of 2000 faults were injected into each program. Again, to magnify the impact of the errors we only inject faults to the 8 32-bit integer registers and the status flags EFLAGS. The difference is that since we have proposed an optimization on safe-register platforms, we shall model that platform in our evaluation.

When we assume that the register file is not protected in hardware, we mimic the fault distribution by randomly selecting a point in the execution sequence and flipping a random bit in a random register. When we assume that the register file is protected in hardware, we mimic the fault distribution by randomly selecting a dynamic instruction and randomly flipping a bit of its “output”. The output can be in a register or in memory if it has been spilled. Memory load instructions are avoided. We call the first scheme “random fault injection” and the second one “safe register fault injection”. Notice that in practice fault distribution is not uniform, but it is a first order approximation used by previous fault injection approaches [53, 54, 70].

After injecting an error into the binary, the program is run to completion (unless it aborts) and its output is compared to a correct output. Depending on the result the program will be categorized as: (unACE): the bit is unnecessary for Architectural Correct Execution [36]; (Detected): the error is detected by our checking code; (Self-Detected): the error is detected by the program assertions; (SDC): Silent Data Corruption, when the program finishes normally but the produced output is incorrect. (SDC) is the first type of errors we want to prevent.
Then, we also want to avoid (Self-Detected) errors and minimize (Seg Fault), but these faults can be recovered, so they are less harmful.

Figure 6.14-(a) and (b) show the experimental results for random fault injection and safe register fault injection, respectively. On average 72% of the faults appear as (unACE), 3% as (Self-Detected), 19% resulted in (Seg Fault) and 6% are (SDC). Under the safe register scheme (Figure 6.14-(b)) more faults result in SDC (9% over 6%). The reason is that the random scheme is more likely to select a dead register. It is also interesting to notice that gzip and bzip2 have a large fraction of Self-Detected errors (10% and 24%, respectively under the random injection scheme) because the program checks the data consistency after the data is compressed or decompressed. Many other real world programs have programmer-inserted assertions as well. These assertions represent another optimization opportunity where some checks can be removed. However, in this dissertation we did not apply any special optimization to handle the Self-Detected errors.

As expected after the program is replicated (Fr), most (Seg Fault), (Self-Detected) and (SDC) go to the (Detected) category. Also, many unACE errors in the original (O) program appear as (Detected) because they are now detected by the checks added. (Fr) has 4.7% and 1.1% of (Seg Fault) under the random register injection scheme and the safe register injection scheme, respectively. (SDC) errors appear under the random register injection scheme because some faults are injected before the value is used but after is checked. (SDC) errors do not appear under safe register injection scheme. After our optimization, ESoftCheck does not produce more (SDC) or (Self-Detected) errors than (Fr). As for (Seg Fault), under the random fault injection scheme, ESoftCheck generates slightly more than (Fr) (5.5% of ESoftCheck versus 4.7% of Fr). Under the safe register fault injection scheme, the (Seg Fault) for ESoftCheck is 3.0% versus 1.1% of (Fr). Remember
that (Seg Fault) is recoverable by rolling back and re-executing, so these numbers are acceptable.

## 6.5 Conclusion

In this chapter we have presented ESoftCheck, a set of compiler techniques that reduce the overheads of software approaches for fault tolerance by removing non-vital checks. To the best of our knowledge ESoftCheck is the first work that identifies the checks before loads, stores and synchronization instructions that can be removed without sacrificing fault coverage. ESoftCheck also takes into account the location of checkpoints, so that recovery is not compromised. When the register file is hardware-protected ESoftCheck can not only remove many checks but also deallocate replicated registers, reducing register pressure significantly.

As a result, in a system that we call MemUnPolluted (where stores execute correctly and memory is not corrupted with wrong results) our techniques reduce execution time by 27.1% over previous state of the art software checking approaches (overheads are reduced from 116% to 70%). Among the three proposed optimizations, the Safe Register optimization is the one that works the best. The reason is that this optimization can reduce register pressure by deallocating some shadow registers early, and register pressure is the major source of overheads on the X-86 platform where we did our experiments. This optimization requires hardware protection (parity or ECC) on register files. Since most modern high end professors already incorporate such hardware protection, the 27.1% performance speedup can be obtained on these processors without hardware modifications.

If full fault coverage is required and the budget of design cost is tight, then software checking is the only viable solution at the expense of a high performance overhead (116%) for instruction replication and checkings. Our ESoftCheck tech-
Figure 6.14: Fault-detection rates break down (MemUnPolluted)
niques can greatly reduce the overhead to 70%. If full fault coverage is not required, ESoftCheck can be easily combined with some partial replication schemes to achieve higher performance gains.
CHAPTER 7

Future Work: Detecting Only Harmful Errors

Not all transient faults will do harm to programs. For the non-harmful errors, choosing to ignore them may reduce the detection cost on them and cut unnecessary rollback recoveries. So it is very interesting to design an intelligent transient faults detector that would be able to catch only harmful errors.

There have been some previous works on studying the probability of errors being masked at architecture level [24,36]. However, even if an error is manifested in architectural execution status, the error still does not necessarily cause a wrong program final result. This occurs, for instance, when the register holding the faulty value is dynamically dead, or when the computation is such that masks the faulty operand, or the control flow takes a wrong path but later converges and the extra instructions that have been executed do not affect the final results (as our examples in Chapter 4), and so on and so forth. In these scenarios, the program intermediate states may be wrong for a while, but at the end it outputs correct result. We call these errors correct-errors. Our experiments in Section 6.4.3 show that as high as 72% of injected errors are correct-errors. For this section of errors, it is totally safe to ignore the error and just let the program continue and finish. Nothing will be wrong. As a bonus, the time for roll-back recovery is saved. More importantly, if the error detection system is intelligent enough, the cost that was used to detect these “safe” errors can be saved too.

If an error causes the program to follow a wrong execution, but the program results in segmentation fault, the situation is not that bad. Since the operating
system sends a signal, the user can infer that “something wrong may have hap-
pened”. When this type of error occurs, the program can continue execution since
the operating system will detect it. We call these errors \textit{segfault-errors}. Some pre-
vious work also use operating system signals as error detectors. Restore [70] and
SWAT [23] find that some errors are very likely to result in abnormal behavior of
the operating system and they use these abnormal behavior as signs of possible
errors. The technique presented in Section 5 takes advantage of this by remov-
ing checks for memory access addresses with the hope that operating system will
detect the errors. However, using operating system as a fault detector can cause
false-positives, since the non-faulty program may have such abnormal behavior.
Therefore, when an abnormal behavior in the operating system is observed, the
system needs to roll-back and re-execute, to determine whether this behavior is
due to an error. In our framework, abnormal behavior signaled by the operating
system only refers to segmentation faults. Since segmentation faults should be
rare for a correct program, the roll-back and re-execution overhead because of a
false-positive alarms should be small.

According to our error classification (see Section 5.2.3), the remaining errors
are those that cause Silent Data Corruption (SDC). These errors are truly harmful.
We call them \textit{SDC-errors}. Actually, there are not many harmful errors: only 6%
as in our experiments (see Section 6.4.3). If an intelligent fault detector is able
to detect only the SDC-errors, there are two obvious benefits: the system does
not have to roll-back on correct-errors; and the cost that was spent on detecting
correct-errors and segfault-errors can be saved.

However, designing such an intelligent transient fault detector is challenging
because it is very difficult to predict the final result of the program execution when
something unusual has been detected. For the fault detection approaches that do
instruction-level replication (hardware-based or software-based), we need to know
which instructions need to be replicated and which values need to be checked in order to catch harmful errors only. The latter question is more important, because the need-to-replicate instructions can be obtained from the backward slice of the need-to-check value, though there are some detailed issues (such as how to handle loop, procedure calls, memory access, pointers, etc) to find these backward slices. For prediction-based fault detection approaches, we need to know where or when to put the symptom or violation detectors to catch harmful errors only. There have been some previous work discussing where to place violation detectors efficiently. Nakka et al. [37] select variables based on their fan-out since a variable with a larger fan-out is more likely to propagate errors to its users. But it is possible that the error in this variable can be masked by its users, and thus resulting in a correct output; or it is possible that errors occurring on its users cause undetected SDC. Therefore, we do not think that the fan-out criteria is good enough for our goal.

In summary, for an intelligent transient fault detector (no matter which approach it uses), the key is to determine what is the minimum set of variables so that, if errors or symptoms are observed on them, then it is likely that the program will result in SDC. We name them “SDC-indicating variables”. After this set is determined, protection can be applied to these variables via instruction-level replication or prediction-based checking.

Notice that most previous transient fault detectors are designed from an opposite direction. They first decide which protection approach to use, apply the protection, and then measure the fault coverage and performance overhead. Taking the likely-invariant scheme [59] (one of the prediction-based fault detection approaches) for an instance, it first chooses variables that have most likely-invariant property, places likely-invariant detector on them, trains the detectors and then prays their detectors can catch most SDC-errors. In this approach, the chosen
variables are known to have great likely-invariant property, however it is unclear whether they are good at catching SDC-errors. Maybe some variables can likely catch correct-errors, leading to unnecessary rollback and diagnosis. And if most SDC-indicating variables do not have likely-invariant property, this scheme will miss most SDC-errors. Our design follows an opposite direction. We first find the variables that can indicate SDC-errors, and then think of how to protect them, using likely-invariant if applicable or using instruction-level replication or others, depending on the cost. With our design path, we have more flexibility in matching different reliability requirement - we can achieve higher fault coverage than pure likely-invariant scheme or do a balance between reliability and cost.

In the following sections, we will discuss in detail the approach we use to find these “SDC-indicating variables” and build the intelligent transient fault detector.

Section 7.1 discusses the methods we are using and why we choose this approach; Section 7.2 gives detailed framework of our design; Section 7.3 presents our intermediate results; Section 7.4 presents previous work related to this topic; and finally Section 7.5 concludes this chapter and discusses future work on this direction.

7.1 Methodology

7.1.1 Metrics of Selecting SDC-indicating Variables

The ideal situation is that by protecting a set of “SDC-indicating variables” all but only the SDC-errors can be discovered. Here, “protect” a variable means that any error happening/propagating to that variable can be discovered immediately. Unfortunately, such a situation is unlikely to happen. One “SDC-indicating variable” can indicate a certain number of SDC-errors (number s); it is also likely to indi-
cate a certain number of segfault-errors (number $f$), and correct-errors (number $c$); and the protection cost varies. We define $SDC$-coverage, $SDC$-distinguishability and protection-cost for each SDC-indicating variable.

**SDC-coverage** shows how many SDC-errors the SDC-indicating variable can detect, namely $SDC$-coverage = $s/ (total \ number \ of \ SDC\text{-}errors)$.

**SDC-distinguishability** gives a sense of how well this SDC-indicating variable can distinguish SDC-errors from other types of errors. $SDC$-distinguishability = $s/(s + f + c)$.

**Protection-cost** defines the cost to protect this variable. Many factors contribute to the cost, eg. the execution frequency of assignments to the variable, the size of backward slice for this variable (for instruction-level replication detection approach), the locality of the variable (for prediction based detection approach), etc.

For any set of SDC-indicating variables, there is SDC-coverage, SDC- distinguishability and protection-cost. SDC-coverage stands for the functionality (fault coverage) of the fault detection; protection-cost reflects the detection overhead; SDC- distinguishability reflects the efficiency of the detection system. SDC-distinguishability is a new concept we propose. The higher SDC-distinguishability means the least unnecessary roll-back. The entire cost for a fault tolerance system is the sum of the detection cost plus the potential roll-back and diagnosis cost.

$\text{(Entire \ cost \ for \ fault \ tolerance) = (Detection \ cost) + (Number \ of \ errors \ signalled) * (Roll-back \ cost + Diagnosis \ cost)}$

As the soft error rate increases, the second component on the right side is projected to play an important role because the roll-back cost and diagnosis cost is huge. The SDC-distinguishability metric can help to reduce the number of errors signalled and substantially mitigate the impact of the second component.

Our design goal is to find a set of SDC-indicating variables with high SDC-
coverage, high SDC-distinguishability and low protection-cost. There is not an
easy trade-off among them. For example, An variable that may affects outside
(eg. write to shared memory location or access I/O mapped location) must be
SDC-indicating and the SDC-distinguishability is 100% (assuming SDC if errors
propagat to other threads or I/O). But its protection-cost may be very high. For
example, with instruction-level replication approach, if we want to protect all such
variables, the cost would be the same as replicating the whole program.

7.1.2 Our Approach to Find SDC-indicating Variables

To find SDC-indicating variables and predict their SDC-coverage and SDC-
distinguishability, we use an experimental approach. By actually injecting faults
to the program and tracking the error propagation, we will know which affected
variables are SDC-indicating and which are not. We inject a huge number of
erors to a program for avoiding biased results. Experimental fault injection
approaches have been seen on analyzing architectual fault mask probability for
different hardware structures [24,71], and fault mask probability at instruction
level [13]. However, experimental fault injection approaches may be limited by
inputs: SDC-indicating variables, SDC-coverage and SDC-distinguishability for a
program may vary from one program input to another, which makes the results
hard to trust when program inputs change. Actually, many experimental analy-
sis approaches have similar problem. Taking likely-invariant scheme [59] for an
instance, a variable having invariant property with one input may not have the
same invariant property with a different input, such that a fitted likely-invariant
detector may not work when input changes. To mitigate the impact from different
inputs, the best we can do is to feed in different inputs and try to find proper-
ties that always hold. We think that the most application level error maskability
comes from the application itself, and depends little on program inputs.

There is an alternative way to find SDC-indicating variables. One can analyze the error mask probability for every types of operation and then calculate the error mask probability for every instructions following data-flow. SymPLFIED [46] is an example of using model-based analysis. It uses symbolic execution to abstract the state of erroneous values in the program and uses model checking to find errors that would evade detection and lead to program failure. This kind of model-based analysis can be done statically, but its comprehensive analysis has bad scalability. It is said in [46] that the exhaustive search performed by SymPLFIED can be exponential in the number of instructions executed by the program in the worst case. Furthermore, a static model-based analysis has several limitations so that we have to make conservative assumptions and make the analysis results inaccurate. For example, when a data is stored to memory, the data flow loses its track and may not know when and where this data is going to be loaded. Then we have to assume the stored data will not be masked by later instructions and will finally contribute to the program output. In addition, if a conditional branch depends on the input, without knowing input at analysis time, we have to assume execution flow can follow both jump branches and have to miss the mask probability that this conditional branch can provide. Considering all these above limitations, we think experimental fault injection approach would serve our purpose better.

### 7.2 Framework

In this section, we describe in detail how we determine SDC-indicating variables and how we use them to build an efficient intelligent transient fault detector.
7.2.1 Fault Injection

Similar to the fault injection experiment described in Section 5.2.3 and Section 6.4.3, we use Pin [26] and inject faults to the binary code excluding system libraries. For each benchmark run, we randomly select a dynamic instruction and flip a bit in its output. Since our experiments are performed on a X86 platform, an instruction output can have different types: if the output is a normal register we change the register value; if the output value is spilled to memory we change the memory value; if the output is only a status flag (such as for “cmp” instruction) we change the status flag. After the fault is injected, the program is run to completion (unless it aborts) and its output is compared to the correct output from a golden-run. Depending on the result the error is categorized as correct-error, segfault-error and SDC-error. Notice that this classification is a little bit different from the the error classification in Section 5.2.3 and Section 6.4.3 for simplification purpose. Here, correct-error is the same as unACE; SDC-error is the same as SDC category; but segfault-error means the error can be noticed by operating system, including SegFault, Self-Detected (program assertion failure) and infinite running. To avoid biased results, we inject totally 30,000 faults to each benchmark.

Some previous work inject faults to microarchitectural level, such as Soft-Arch [24], ReStore [70] and others. In this way, the injected fault rate and distribution is relatively accurate, however many injected faults would be masked by the microarchitecture and only a few would manifest as errors in the architectural status. Since our work focuses on fault maskability by applications, we are modeling only those errors that appear in the architectural status. In this way running experiments on a real machine, our fault injection is very fast so that we are able to run a program to completion seeing the final output and a large number of
runs are possible.

Our fault injection implementation misses two types of errors: errors in instruction opcode and errors in program counter. It could happen that the instruction decoder is affected by a fault and mistake one opcode to another: if an computation opcode is mistaken to another computation opcode, we may mimic the effects by injecting multi-bit errors to the instruction output; if an computation opcode is mistaken to a store or a store is mistaken to a non-store opcode, the data chain through memory will be corrupted and it is hard to mimic the effect. An error in program counter may drive the execution flow jump to a random illegal position, and cause later instructions operate on wrong source values and generate wrong outputs. We think it is difficult to correlate the mask probability of the errors in PC with application properties (instruction type or source code structure or others), so that we would leave the errors in PC to be protected by a cheap separate mechanism (refer to the discussion in Section 3.3-III for previous proposed mechanisms) and don’t include these errors here.

7.2.2 Tracking Error Propagation

A soft error in the architectural execution status may change the correct control flow, may corrupt the values of variables in register or memory, may propagate to outside (e.g. I/O). Because a harmful error in control flow will finally corrupt values of variables or propagate to outside (otherwise the error does not affect program final output at all), we only monitor values in variables and values that are going to affect outside world. For the same reason, since SDC-errors must manifest themselves in stores or system calls, we track only operands of memory stores and arguments of system calls. Notice that if a register value is spilled to memory, we don’t treat it as a memory store. For stores to dynamically allocated
heap variables, since the addresses may vary from one run to another, we don’t record these addresses. To help to identify which variable is affected by the error, we also record the instruction PC along with the its operands. Due to space limitations, we take a record of 9,999 memory stores and system calls after injecting a fault, and then stop the tracking and let the program complete. We call the record of these tracking entries a “value trace”.

Sometimes if a fault corrupts the higher bits of the address of a memory access instruction, we receive a segmentation fault immediately. In this case we may record just a few store/system call entries or have no value trace at all.

If one fault injection run generates a complete value trace (there is no segmentation fault before the tracking completes), we compare the trace with the correct records from a golden run. The fault-injected value trace can be exactly the same as the records from the golden-run, either because the the fault has been masked before propagating to memory stores and system calls, or because the error is hidden so deep that it manifests later than 9,999 stores and system calls. The fault-injected value trace can show some errors in operands of stores or arguments of systems calls, or mismatch in instruction PCs meaning corrupted control flows. Figure 7.1 gives two examples of fault-injected value traces (Figure 7.1-(b)(c)) and the corresponding value trace from the golden-run (Figure 7.1-(a)). In these examples, each line represents a tracking record in a format as follows.

[Serial Numver] [PC of the tracked instruction]:
[the values to be tracked]

In the fault-injected value trace, the serial number in front of every line shows how far this entry is from where the fault is injected. Here we assume that a fault-injected value trace keeps only 7 trace records.

In the real implementation, sometimes there are multiple values behind a PC.
Figure 7.1: Examples of fault-injected value traces and the corresponding golden-run value trace. The items in bold italic are found different from the golden-run.

For example, we may track both value and address for a memory store. In our examples here, we only show one value for each tracked instruction for simplicity purpose.

Figure 7.1-(b) shows an example of fault-injected value trace where only variable values are different from the golden-run value trace. In Figure 7.1-(c), the PC starts to mismatch from Line 2, meaning the control flow goes wrong before it reaches the pc4 instruction.

### 7.2.3 Finding SDC-indicating Variables

To better see which variables (or instructions that output the variables) are vulnerable to faults, we sort the entries in a fault-injected value trace by instruction (PC). The recorded values for the same instruction are placed in the same order as they appear in the original value trace. Along with the recorded values, we tag their serial numbers together. After the sort, we compare the recorded values to those from the golden-run for each tracked instruction. Once a mismatch is first found at value $V$, we stop the comparison for that instruction and flags the serial number $S(V)$ tagged with value $V$. Put in another words, pretending there is
a perfect value predictor for a tracked instruction $I$, this perfect value predictor can catch the error at the position where the $S(V)$-th store/system call is made. When control flow is corrupted, the comparison becomes a little complicated.

We use two examples (Figure 7.2-(b)(c)) to show how to sort fault-injected value trace by instruction PCs and compare them to corresponding golden-run value trace (Figure 7.2-(a)). In the fault-injected value trace shown in Figure 7.2-(b), where only values are different from the golden-run value trace, after the sorting and comparison, the value records for pc1 and pc3 are exactly the same as the records from the golden-run (we put “pass” for pc1 and pc3) and because the first mismatched value record for pc2 is “$80<2>$”, we put “wrong-value$<2>$” for pc2. “$<2>$” is the position for pc2. It means if we have a perfect value predictor for instruction pc2, we can catch this error when the 2nd memory store/system call is executed after the error is injected. The example shown in Figure 7.2-(c) is a little complicated, because the control flow is corrupted and the instruction pc5 is executed in a place where it should not be. As a result, the tracking records from pc2 and pc3 are not in their correct positions. After the sorting, we find that the value records for pc2 are still the same as the records from the golden-run but the tagged serial number starts not to match at position 5 (“wrong-order$<5>$”); pc3 misses a value record in the end (“miss”), and the serial number at position 6 does not match (“wrong-order$<6>$”); and pc5 has an extra value record at position 4 (“extra$<4>$”). If an instruction has more than one symptoms at the same time, such as pc3 having “miss” and “wrong-order$<6>$”, we only record one symptom that is the most important. In our definition, a symptom with smaller position is considered more important. For instructions having “wrong-order”, “miss” and “extra” symptoms, even if we have a perfect value predictor, we may not be able to catch the error within the limited detection period. We call these control-flow-based symptoms. Here, the word “symptom” refers to how a variables is impacted.
When the sorting and comparison is done, we can easily see which variables (instructions that produce the variables) are likely to be affected by injected faults and how they are affected, in variables values or by wrong control flow.

Considering all injected faults with which the value traces are complete, we count how many faults get the symptoms of “wrong-value”, “wrong-order”, “miss” and “extra” for each tracked variable (PC). Table 7.1 shows the table headings that we use for this fault impact statistics. The table is called variable-symptom table.

Because we know the final program result for each injected fault, we can do the same fault impact statistics for only SDC-errors. Now we have one variable-symptom table for all injected faults (excluding those without complete value traces) and one variable-symptom table for SDC-errors. If there is a perfect symptom predictor, we can catch errors that show such a symptom. Thus, the numbers in the variable-symptom table for SDC-errors reflect SDC-coverage, and the ratio of the numbers in the variable-symptom table for SDC-errors over the corresponding numbers in the variable-symptom table for all faults mean SDC-distinguishability. Therefore, we can choose SDC-indicating variables based on the statistical numbers of SDC-coverage and SDC-distinguishability.

<table>
<thead>
<tr>
<th>PC</th>
<th>total</th>
<th>wrong-value</th>
<th>wrong-order</th>
<th>miss</th>
<th>extra</th>
</tr>
</thead>
</table>

Table 7.1: The variable-symptom table headings. Every cubic in the table shows how many times a variable behaves a particular symptom from errors.
Figure 7.2: Examples of sorting fault-injected value traces by instruction PCs, and comparing the sorted value traces to corresponding golden-run value trace.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Golden-Run Value Trace</th>
<th>Fault-Injected Value Trace 1</th>
<th>Fault-Injected Value Trace 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 pc1: 128</td>
<td>pc1: 128</td>
<td>pc1: 51214</td>
<td></td>
</tr>
<tr>
<td>2 pc2: 0</td>
<td>pc2: 80</td>
<td>pc2: 0</td>
<td></td>
</tr>
<tr>
<td>3 pc3: 86</td>
<td>pc3: 86</td>
<td>pc3: 86</td>
<td></td>
</tr>
<tr>
<td>4 pc2: 2</td>
<td>pc2: 82</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5 pc3: 85</td>
<td>pc3: 85</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6 pc2: 4</td>
<td>pc2: 84</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7 pc3: 79</td>
<td>pc3: 79</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Sorted Value Trace 1</th>
<th>Sorted Value Trace 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>pc1: 128&lt;1&gt;</td>
<td>pc1: 128&lt;1&gt;</td>
<td>pc1:51214&lt;1&gt;</td>
</tr>
<tr>
<td>pc2: 0&lt;2&gt;,2&lt;4&gt;,4&lt;6&gt;</td>
<td>pc2: 0&lt;2&gt;,2&lt;5&gt;,4&lt;7&gt;</td>
<td>pc2:0&lt;2&gt;,2&lt;5&gt;,4&lt;7&gt;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Comparison</th>
</tr>
</thead>
<tbody>
<tr>
<td>pc1: pass</td>
<td>pc1: wrong-value&lt;1&gt;</td>
</tr>
<tr>
<td>pc2: wrong-value&lt;2&gt;</td>
<td>pc2: wrong-order&lt;5&gt;</td>
</tr>
<tr>
<td>pc3: pass</td>
<td>pc3: wrong-order&lt;6&gt;</td>
</tr>
<tr>
<td>pc5: extra&lt;4&gt;</td>
<td>pc5: extra&lt;4&gt;</td>
</tr>
</tbody>
</table>

(b)golden-run (c)fault-injected (sorting) (comparing to golden-run value trace)
7.2.4 Building Real World Intelligent Error Detector

In the previous subsection, we assume a perfect symptom predictor to catch errors that cause such symptoms as “wrong-value”, “wrong-order”, “miss”, “extra”. However, if we consider the cost, it is impractical to implement such perfect symptom predictors in hardware or software. As a result, we can not simply choose to protect those variables that have high SDC-coverage and SDC-distinguishability. We have to take into account how much effort it is required to protect each variable. In this subsection, we describe how we make use of the statistics calculated in the previous subsection to build an intelligent efficient error detector. We consider the whole detector a combination of separate detection mechanisms.

First, we pick the variables whose SDC-distinguishability exceed a certain threshold. We would first try 80%. If we are not satisfied with this number, we can adjust it later. Notice that SDC-distinguishability is calculated for every pair of variable (PC) and symptom.

For every chosen pair of variable and symptom, we try to find “cheap” protection mechanism and evaluate the cost for that pair.

I If the symptom is value-based, namely “wrong-value”, we will profile the program and see how easy to build a likely-invariant checker [59] or a perturbation-based screener [49] or other kinds of value predictors for this variable (PC). If nothing fits at all, we estimate the effort to replicate the instructions in the backward slice for this variable.

II If the symptom is control-flow-based, namely “wrong-order”, “miss”, “extra”, we trace back from the chosen variable and try to find the place that makes the control flow go wrong. Notice that in our tracking framework, we only track variables that may go to memory or outside. There are much more internal variables we don’t track, but they can be SDC-indicating and
cheap to protect. For example, there must be some internal variables in
the error propagation path that steer the control flow to wrong directions
and finally cause control-flow-based symptoms on the outgoing variables.
We can not track all internal variables due to time and space limit. What
we do is we track outgoing variables and once we see interesting control-
flow-based symptoms on the tracked variables we trace back to find those
critical internal variables. Then we can use the method mentioned in I for
the “wrong-value” symptom to protect these critical internal variables.

It could happen that there is no cheap protection at all for a pair of variable
and symptom. Then we have give up protecting that pair. Notice that one pair
may have overlap with another pair in terms of SDC-coverage. We don’t have
to design separate protection mechanism for every pair of variable and symptom.
But we need to examine the overlap to minimize the protection efforts.

Down to the real world, the statistical SDC-coverage and SDC-distinguishablity
is just an upper bound. It is expected that the designed likely-invariance checker
or the perturbation-based screener do not catch all errors and may sometimes pro-
duce false positives. In this case, with the profile knowledge, we need to update
the SDC-coverage into a closer number that we can achieve.

If the entire SDC-coverage from all the pairs that we can protect is not enough,
we need to relax the threshold on SDC-distinguishability and choose more pairs
of variable and go back to the first step.

7.2.5 Bounding Fault Detection Latency

In our intelligent fault detection system, we assume a underneath checkpoint layer
to support roll-back recovery whenever an error is detected or suspected. To make
sure the checkpoint-recovery layer is able to recover properly in case of error, we
must bound the latency between the fault occurring and the fault being detected.

Our fault detection framework is very natural in bounding fault detection latency. Recall that after the sorting and comparison, we record for each faulty instruction not only the symptom it behaves due to the error, but also the position the symptom occurs. The position is the number inside the “< >” in Figure 7.2, telling how far (in terms of the number of memory stores/system calls) the symptom occurs since the error is injected. Given a maximum detection latency, if we can catch all symptoms whose positions are within the bound, we are doing the best we can. Therefore, if we count only variable-symptom pairs whose positions are within the given bound when we build the variable-symptom table, the priority for choosing which variable-symptom pairs to protect will be given to symptoms that are close to where the error first manifests in architectural status.

Notice that when our fault detector catches a symptom that occurs within the given detection latency, the detector is still capable to catch the same symptom that happens later than the given detection latency (meaning that the causing error hasn’t been caught elsewhere before). In this case, although the underneath checkpoint layer is not able to do correct roll-back recovery, it is still better to report the error than just letting the harmful error sneak away.

Besides SDC-errors, we also have to consider bounding fault detection latency for segfault-errors. Although in our primitive design we intentionally leave segfault-errors to the operating system for protection, it is possible that operating system reacts too late so that the checkpoint recovery layer can not do proper roll-back. We can tell these “late reaction” by looking at where the program crashes (segmentation fault or failing program assertion) from the value trace point of view. For example, if the program crashes after the 9,999 tracking entries complete, the detection latency for this error caught by operating system is longer than 9,999 memory stores/system calls. We must treat late reaction segfault-
errors the same as SDC-errors, in order to properly bound the detection latency for all errors.

The last question is whether the number of memory stores/system calls is a proper unit to count the detection latency. Some previous work use the number of dynamic instructions to count the detection latency [23, 49, 70]. Those papers suggest a pipeline flush to implement the recovery, so that the detection latency must be bounded by the length of the pipeline, which is counted in the number of instructions. In our work, we assume a software/hardware checkpoint and recovery layer that is relatively cheap and can sustain long detection latency. (Although the recovery expense is very high, the impact to overall performance is little if the error rate and the false positive rate is small.) Such a software/hardware checkpoint and recovery layer makes snapshot of architectural status, records lists of the memory data that differ in the checkpoint, and holds the values that may go outside. Therefore, how much detection latency the checkpoint and recovery layer can support depends on the number of memory updates and I/Os it can hold. This unit is just what we use to bound the detection latency in our fault detector design.

7.3 Intermediate Results

In this section, we present preliminary results from our experiments on two SPECINT2000 benchmarks, gzip and twolf.

7.3.1 About Application Fault Mask Probability

We performed our fault injection experiments as described in Section 7.2 and injected 30,000 faults to each benchmark. The fault injection results show that the fraction of correct-errors, segfault-errors and SDC-errors are consistent with
our results from previous fault injection experiments presented in Section 6.4.3. Only 2.6% errors in gzip and 31.1% errors in twolf cause Silent Data Corruption, the most harmful errors we want to prevent. And 62.5% errors in gzip and 62.4% errors in twolf do not corrupt program output at all. Thus, we can consider that 62.5% and 62.4% are the fault mask probability of the application.

Instruction and control-flow error derating.

Figure 7.3 shows detailed impacts on our tracked value traces from injected faults. In our experiments, for each injected fault, we track 9,999 memory stores and system calls after the fault is injected. If the tracking completes and all entries in the resulting value trace match the records from the golden-run, it means the fault does not propagate to memory stores or system calls at all (faults are masked by some instructions or control flow jumps), or the fault is hidden too deep so that it does not manifest in our trace within 9,999 entries. Actually, we observe a few deep-hidden faults in experiments on gzip, where a loop induction variable is corrupted. The loop induction variable contributes only to counting loop iterations. So after the induction variable becomes incorrect, everything else
inside the loop is still correct – we see correct value trace, until the loop is to finish where our tracking has already stopped before. Fortunately, these deep-hidden faults at last fail program assertions and attract our attention. An error is *derated* if it does not manifest in the final program output. If we just pretend a 9,999-entry value trace is long enough to catch all manifested errors, the fraction of “matched complete value traces” represents the probability of error derating from instructions and control-flow jumps. Figure 7.3 shows that the instruction and control-flow derating rate is 52.9% for gzip, 46.7% for twolf. Cook et. al show that the average instruction-level derating rate on SPECINT2000 benchmarks is 35.9% [13], lower than our numbers here. There are two reasons for this difference: 1) Our numbers include control flow derating. If a conditional branch is affected by an error and goes wrong direction but no memory writes or system calls are affected by this fault, we count the case in the derating rate, which is not by Cook. 2) In their experiments, faults are also injected to instruction opcode which we don’t do. The faults in instruction opcode usually can not be masked by instructions. So their overall error derating rate is lower than ours.

**Application error derating.** Our experiments show more error derating opportunity than just instruction and control-flow error derating: for gzip 9.64% errors corrupt value trace but in the end don’t corrupt the final program output; for twolf, the number is 15.7%. Furthermore, for those segfault-errors that corrupt value trace (24.3% for gzip) we can leave operating system to detect them. After all, the most harmful errors are those causing SDC.

### 7.3.2 Exploring SDC-indicating Variables

**gzip**

For gzip, we totally track 342 out-going variables (outputs of memory stores/system
calls) all over the whole program. Among them, 261 variables have symptoms for all injected faults, and 145 variables have symptoms for SDC-errors. Following the instructions in Section 7.2.4, we calculate SDC-distinguishability and SDC-coverage for every pair of variable and symptom that SDC-errors cause. After setting the threshold for SDC-distinguishability as 80%, we choose 101 variable-symptom pairs, including 30 value-based symptom pairs and 71 control-flow-based symptom pairs.

Since control-flow-based symptom pairs take a large portion, we investigate the program code and try to find internal values that are in the error propagation path and lead to the symptom happening. We find that 339 SDC-errors propagate to the return value of function \( ct_tally() \), which drives the control flow into calling \( flush_block() \) earlier or later than it should. The good thing is that \( ct_tally() \) should return a boolean value, 1 or 0. A very simple range detector for the return value on the caller side can catch most of these errors if the return value is found to be not 1 or 0. From our profile, 328 SDC-errors can be detected by this simple range detector and there is no false positive at all, meaning 42.3% SDC-coverage, 100% SDC-distinguishability and near zero protection-cost. If we want to cover more SDC-errors, we may need to relax the threshold for SDC-distinguishability, which we haven’t done.

**twolf**

For twolf, we track 1342 outgoing variables through out the whole program. Among them, 682 variables have symptoms for all injected faults and 635 variables have symptoms for SDC-errors. After setting the threshold for SDC-distinguishability as 80%, we choose 1574 variable-symptom pairs, including 1005 value-based symptom pairs and 569 control-flow-based symptom pairs.

Since value-based symptom pairs take a large portion, we can just look at the chosen variables and try to build cheap error detection mechanisms for them. We
Table 7.2: Three chosen variables to protect in twolf

<table>
<thead>
<tr>
<th>variable</th>
<th>function</th>
<th>SDC-coverage</th>
<th>SDC-distinguishability</th>
</tr>
</thead>
<tbody>
<tr>
<td>var1</td>
<td>new_dbox()</td>
<td>90.6%</td>
<td>93.8%</td>
</tr>
<tr>
<td>var2</td>
<td>new_dbox_a()</td>
<td>94.3%</td>
<td>88.6%</td>
</tr>
<tr>
<td>var3</td>
<td>new_dbox_a()</td>
<td>88.5%</td>
<td>92.0%</td>
</tr>
</tbody>
</table>

Table 7.3: Range profiles for the chosen three chosen variables

<table>
<thead>
<tr>
<th>variable</th>
<th>value range</th>
<th>stride range</th>
</tr>
</thead>
<tbody>
<tr>
<td>var1</td>
<td>2202-5892</td>
<td>(-1173)-806</td>
</tr>
<tr>
<td>var2</td>
<td>2206-5993</td>
<td>(-976)-690</td>
</tr>
<tr>
<td>var3</td>
<td>4794967292-4294967294</td>
<td>(-2)-2</td>
</tr>
</tbody>
</table>

don’t do a thorough study on all chosen variables. We randomly choose three variable-symptom pair which have relatively high SDC-coverage and high SDC-distinguishability shown in Table 7.2.

From the table, we can see that if there is a perfect value predictor for any one of the three variables, we can detect around 90% SDC-errors and pay little expense on unnecessary rollback and recovery. However, such a perfect value predictor do not come for free. In our experiment, we place a very simple range detector for each of the variables. The range detector monitors two properties for a variable: the value of the variable, and the stride from the previous value to the current value of this variable. If the value or the stride violates the profiled range, the range detector will trigger an alarm. The performance overhead for such a range detector on these three variables is 1.3% over the non-protected program. The last two columns in Table 7.3 show the two ranges for each variable from the profile of twolf. We turn on the simple range detectors and do fault injection. We inject the same 30,000 faults as we did, and find that the simple range detectors can catch 3928 SDC-errors (42.0% SDC-coverage), 2 segfault-errors and only 580 correct-errors (87.1% SDC-distinguishability). Notice that there are 4716 correct-errors having mismatched value traces (see Figure 7.3), all these 4716 correct-errors
could cause false positives. We avoid most of them. From the Table 7.3, it can be seen that we don’t intentionally pick variables with narrow ranges. If strictly following instructions given in Section 7.2.4, we should profile all variables from 1005 value-based symptoms that satisfy SDC-distinguishability threshold and pick variables with narrow ranges and high SDC-coverage. In this way, we will surely achieve higher SDC-coverage and suffer less false positives.

7.3.3 Measuring Fault Detection Latency

In our preliminary experiments, we do not follow what we have discussed in Section 7.2.5 to bound the detection latency for detected errors. Now we just measure the fault detection latency for the systems we construct in last section, in which only a simple range protector is applied to the chosen variables. The purpose of this measurement is to see how difficult it is to bound the detection latency. If a range protector triggers an alarm, we measure the distance between the instruction where the fault is injected and instruction where the alarm is triggered. If the system triggers an alarm, in a form of segmentation fault or assertion failure, we also measure the distance between where the fault is injected and where the alarm is triggered. Figure 7.4 shows the results. Every benchmark has three bars, representing the percentage of segfault-errors, SDC-errors and correct-errors over the total number injected errors, respectively. For each bar, errors are classified into eight categories based on how soon they can be detected or can not be detected at all. The “no trace” category means the the system triggers an alarm very quickly, before the execution reaches a tracked instruction. A tracked instruction is a memory store or a system call instruction.

The results show that for gzip, all detected SDC-errors can be bounded within 5 tracked instructions. However, only 31.2% of detected segfault-errors can be
bounded within 10,000 tracked instructions. The reason is that the segfault-errors here include not only 3374 errors that cause segmentation fault, but also 7084 errors that fail programmer assertions. For the latter section of errors, the errors are hidden so deep that it usually takes very long for programmer assertions to detect them. Therefore, if we want to bound error detection latency, we should treat these segfault-errors with long detection latency as SDC-errors and find the proper variables to protect them.

For twolf, the results show that 98.7% of detected segfault-errors occur immediately after the fault is injected, before any instruction is recorded. 82.6% of detected SDC-errors can be bounded within 5 tracked instructions and 98.1% within 100 tracked instructions. All detected SDC-errors can be detected within 9,999 tracked instructions. In short, this constructed fault detector (simple range protector on these chosen variables) can detect 88.1% of detected segfault-errors and detected SDC-errors within 5 tracked instructions, and detect 98.6% within 100 tracked instructions.
7.4 Related Previous Work

We compare this work to previous work in two aspects.

7.4.1 Fault Masking

When the hardware is attacked by alpha particles or energy neutrons, the fault may be masked at micro-architecture level, architecture level, instruction level or application level. Mukherjee et al. [36] and Li et al. [24] study the error mask probability on some structures at architectural-level and use experimental method to predict the architectural error derating. Several works are proposed to ignore errors that will be masked by architecture or instructions. Weaver et al. [73] modifies the error reporting mechanism and not to signal detected errors that only affect dynamically dead instructions. Cook et al. [13] observe six categories of instruction level error derating and propose in DMR system to check two register status periodically so that the errors that can be masked by instructions will not be detected. In this chapter, we focus on instruction level and application level fault masking and we are the first to propose an intelligent fault detector that is able to only detect errors that are not to be masked by instructions and the application.

7.4.2 Invariance-based Fault Detection

The invariance-based fault detection mechanism is a subclass of prediction-based fault detection approaches (refer to Section 2.2.1). Because value can have localities, a profile of expected values can be established from historical information or current processor state. When current value is found outside of the profile, a warning will be indicated. There have been some work using value invariance violation detector to find transient errors. Racunas et al. [49] uses value perturbation
screeners to detect hardware transient errors. Sahoo et. al [59] find variables that have likely invariance property from training runs, and place invariance detector on the field to catch hardware permanent errors. Our work is different from the above invariance-based fault detection mechanisms in two aspects: i) they choose to monitor variables that have most locality property. It is true that monitoring these variables will get low false positives. But it is possible that the caught errors (if they are not false positives) will be masked by later instructions or the application. In our work, we choose variables firstly based on their capability to indicate SDC-errors. 2) we use invariance-based fault detection mechanisms to protect the chosen variables, but we are not restricted to these mechanisms. We first find the variables that can indicate SDC-errors, and then think of how to protect them, using likely-invariants if applicable or using instruction-level replication or others, depending on the cost. Therefore, we have larger space to play in SDC coverage. In the case that all variables in the program have little invariance property, the pure invariance-based detection mechanism will fail catching SDC-errors. While we can choose instruction-level replication to protect chosen SDC-indicating variables.

7.5 Conclusions and Future Work

This chapter presents an interesting research direction on designing an intelligent fault detector that is able to catch only harmful errors to a program. If a fault detector can automatically ignore errors that will be ultimately masked by the application we can save unnecessary roll-back recovery and the cost spent on detecting the non-harmful errors. The most harmful errors to a program are errors that will cause Silent Data Corruption. The key task to design an intelligent fault detector is to determine a set of SDC-indicating variables. By protecting them,
most SDC-errors but few other types of errors can be caught (high SDC-coverage and high SDC-distinguishability). At the same time we need to consider the fault detection mechanisms for these variables (low protection-cost) and bring the protection cost into the trade off between fault coverage and performance.

We use experimental methods to find the set of SDC-indicating variables to protect. By injecting and tracking a lot of architectural execution errors, we get a statistical knowledge of which variables and what symptoms on these variables are most likely to suggest SDC-errors. Then these variables and their symptoms can be chosen to protect. If the symptom is value-based, we can try invariant-based error detection mechanisms to protect them; if the symptom is control-flow based, we look into the code and find some internal variables that are on the error propagation path and cause such control-flow based symptoms. Then we can use invariant-based error detection mechanisms to protect the internal variables. The final variables chosen to protect depends on their SDC-coverage, SDC-distinguishability and their protection-cost. We also discuss how to bound the error detection latency when choosing SDC-indicating variables.

We did preliminary experiments on two SpecINT2000 benchmarks, gzip and twolf. For gzip, we find that with a very simple 0-1 value checker on a function return value, we can detect 42.3% SDC-errors without any false positives (errors that will be ultimately masked). For twolf, we randomly choose 3 variables with high SDC-coverage, high SDC-distinguishability. By applying simple range violation detectors on them, we can catch 42.0% SDC-errors, and only 12.3% of all false positives.

In the future, we have the following tasks to do on this direction.

I Do experiments on more benchmarks to show the efficiency and intelligence of our fault detector design.
II For control-flow-based symptoms, we need to find the internal variables that on the error propagation path and do not mask the error. Currently, we do this by manually investigating into the code. We need to automate the process. For value-based symptoms, we need to profile the invariance properties on them and evaluate the protection efficiency. In our experiments, we just try simple range violation checkers. We can try more invariant properties and automate the process.

III On the stage when we know which variables and what symptoms are likely to suggest SDC-errors and how much protection cost they may cause, it is still not easy to choose the best set of variables and symptoms that can achieve high SDC-coverage, high SDC-distinguishability and low protection cost. There are many challenges here: different variables may have overlap on errors that they can indicate; an invariant-based error detector can not catch all errors on the protected variables and may introduce false positives, which makes the statistical SDC-coverage and SDC-distinguishability not accurate; the protection cost of replicating a backward slice is hard to predict; it is very hard to do trade-offs between the three metrics; etc. On this chapter, we just list all the issues we have to consider when we choose which variables to protect and propose an incremental approach on the choosing process. We need a further study on it.
CHAPTER 8

Conclusions

In this dissertation, we propose three techniques for efficient software checking. First, we identify a code pattern that corresponds to outcome tolerant branches, and develop a compiler algorithm that finds these patterns, avoiding unnecessary replication and checking. Second, we evaluate the removal of address checks for loads and stores, and analyze situations where these checks can be removed with little loss of fault coverage. We also identify the check and replicated registers that can be removed on a register safe platform. Third, we present ESoftCheck, a set of compiler techniques that reduce the overheads of software approaches for fault tolerance. To the best of our knowledge ESoftCheck is the first work that identifies the checks before loads, stores and synchronization instructions that can be removed without sacrificing fault coverage. ESoftCheck also takes into account the location of checkpoints, so that recovery is not compromised. When the register file is hardware-protected ESoftCheck can not only remove many checks but also deallocate replicated registers, reducing register pressure significantly. We also discuss our future work to find SDC-indicating variables and design an intelligent fault detector that can most likely detect only harmful errors, so that for the errors that will be masked by instructions and applications, the cost spent on detecting them and unnecessary rollbacks can be saved.

Our proposed techniques can be applied to meet different reliability and performance requirements:

If full fault coverage is required and no hardware support is provided, we have
to use software checking approach to replicate instructions and pay high performance overhead (116% over the original non-protected code). Then we can apply ESoftCheck framework to remove non-vital checks and decrease the performance overhead to 70%. If the application has many short-cut paths, then we can apply the boolean logic optimization to further improve performance without losing fault coverage. Notice that not all applications require such a high fault coverage, such as cellphone applications, entertainment applications, etc. However, there are still some applications that need almost full fault coverage. If hardware modification is impractical in these environment, we have to pay the 70% performance overhead to meet the reliability goal.

If the requirement for the fault coverage is high but not as high as the first case, we can do the following as the fault coverage requirement decreases. 1) Adjust the knob of degree of trustiness in ESoftCheck to remove more non-vital checks. 2) Apply the technique of removing address checks before loads. 3) Apply the technique of removing address checks before loads and stores. When we apply the above techniques, some of vital checks may be removed and some errors may cause SDC without being detected (1.44% SDC if applying the removal of address checks before loads and stores alone). It also could happen that the system can not properly recover when an error is detected, because the error detection latency exceeds the capacity that the underneath checkpoint-rollback layer can support. If the removal of address checks before loads and stores is applied alone on top of the baseline fully replicated software checking, it improves the performance by 24.8%, but increases SDC rate to 1.44%. And 4.6% of the detected errors (equivalent to 2.1% of the total injected errors) can not be detected within 100,000 instructions. If the removal of address checks are applied on top of ESoftCheck, we expect to lower the performance overhead below 70%. And the 1.44% SDC is quite acceptable because the real probability of the SDC rate that we can observe is
1.44% multiplied by the raw SER rate and the probability of errors propagated to architectural states, which are small numbers. The number of 2.1% for unbounded errors is acceptable for the same reason.

If the requirements decrease even more, we can selectively protect some modules in an application or provide some special hardware support if possible (eg. load value queue, store value queue, hardware checks, etc). Our three techniques can still be applied on top of it to reduce performance overhead.

If the budget on performance overhead is very tight, we can do symptom-based software checking for the application. Then the technique that has been discussed as our future work, detecting only harmful errors, can help to increase the system efficiency and flexibility by only placing the limited resources to the key places, where the variables that are monitored by the fault detection system are most likely to detect harmful errors. Our preliminary results show that we can detect more than 40% SDC errors with little performance overhead (negligible overhead for gzip and 1.3% overhead for twolf).

In a summary, software checking approaches are very promising in the near future when the impact of transient faults on commodity processors becomes serious. Software checking approaches need little hardware modification and can be configured to meet different reliability requirements. Although the performance overhead is a little bit high compared to hardware-based approaches, in the situations where the budget is tight one has to pay the price on hardware design or on performance if high reliability is required. For software checking approaches, the price can be only paid to applications that need good reliability so that the casual applications running on the same machine are not affected. In order to make software checking approaches more attractive, we have proposed three techniques and discussed one future work that can greatly reduce the performance overhead of software checking approaches and maintain a similar reliability. Our techniques
can be combined with other varied software checking techniques to meet different requirements for reliability and performance.
REFERENCES


Fault-tolerance design of the ibm enterprise system/9000 type 9021 processors.


Author’s Biography

Jing Yu was born in Wuhan, China in 1979. She received her bachelor degree in Computer Science from China University of Science and Technology in June of 2003. Since 2003 she has been a research assistant in the Computer Science Department at University of Illinois at Urbana-Champaign. Her research interests include computer architecture and compiler.