SOFTWARE MODULATED FAULT TOLERANCE

GEORGE A. REIS III

A DISSERTATION
PRESENTED TO THE FACULTY
OF PRINCETON UNIVERSITY
IN CANDIDACY FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY

RECOMMENDED FOR ACCEPTANCE
BY THE DEPARTMENT OF
ELECTRICAL ENGINEERING

ADVISER: PROFESSOR DAVID I. AUGUST

JUNE 2008

Abstract

In recent decades, microprocessor performance has been increasing exponentially, due in large part to smaller and faster transistors enabled by improved fabrication technology. While such transistors yield performance enhancements, their smaller size and sheer number make chips more susceptible to transient faults. Designers frequently introduce redundant hardware or software to detect these faults because process and material advances are often insufficient to mitigate their effect. Regardless of the methods used for adding reliability, these techniques incur significant performance penalties because they uniformly protect the entire application. They do not consider the varying resilience to transient faults of different program regions. This uniform protection leads to wasted resources that reduce performance and/or increase cost.

To maximize fault coverage while minimizing the performance impact, this dissertation takes advantage of the key insights that not all faults in an unprotected application will cause an incorrect answer and not all parts of the program respond the same way to reliability techniques. First, this dissertation demonstrates the varying vulnerability and performance responses of an application and identifies regions which are most susceptible to faults as well as those which are inexpensive to protect. Second, this dissertation advocates the use of software and hybrid approaches to fault tolerance to enable the synergy of high-level information with specific redundancy techniques. Third, this dissertation demonstrates how to exploit this non-uniformity via Software Modulated Fault Tolerance. Software Modulated Fault Tolerance leverages reliability and performance information at a high level and directs the reliability choices at fine granularities to provide the most efficient use of processor resources for an application.

Acknowledgments

First, I thank my advisor David August for his support during my graduate school tenure. He allowed me to pursue research that I felt was important although it was a departure from his primary interests, and I am very grateful for that freedom. His probing and insightful discussions regarding all aspects of this research including design, implementation, and presentation were instrumental in refining this dissertation. I thank Matthew Bridges, Easwaran Raman, Ram Rangan, Guilherme Ottoni, Jason Blome, Adam Stoler, and the rest of the Liberty Research Group for their input on various aspects of this dissertation. In particular, I thank David Penry for his work on the IA-64 LSE model as well as Spyridon Triantafyllis and Matthew Bridges for their work on the Velocity compiler. Both of these pieces of infrastructure were essential components in the evaluation of my implementations.

The ideas presented in this dissertation evolved and were refined through interactions with many individuals. In particular, I would like to express my gratitude to Neil Vachharajani and Jonathan Chang. They both had essential roles in motivating this research through countless discussions, and I greatly appreciate their enthusiasm and interest in my work that was evident during brainstorming, implementing ideas, writing up research papers, and rehearsing presentations.

My dissertation committee has played an integral role in the development of this work. I thank Shubu Mukherjee for his contributions towards the ideas in my dissertation and overall reliability research program. His guidance as a mentor during my internship at Intel in addition to his industrial and broad impact perspectives have been invaluable in shaping the direction of my research. I also thank Niraj Jha for his thoughtful, detailed comments on my dissertation and for helping me refine the expression of my ideas. I am grateful to David Walker and Li-Shiuan Peh for their insightful comments on my dissertation proposal, and specifically their help in positioning this work in a broader context.

I would like to thank the National Science Foundation (CNS-0627650 and CNS-0615250) and the Intel Corporation for providing material support for this research. In addition, I would like to express my gratitude to the anonymous donor and the Princeton University Graduate School for the Harold W. Dodds Fellowship that supported my last year of research.

Finally, I thank my friends and family for their support throughout graduate school, during the good but especially the hard times. I thank my mother Debra and my father George for their unconditional love, encouragement, and confidence in me. My father was unable to see me graduate, but he instilled in me the importance of hard work and family, and continues to inspire me to be a better man. Above all else I sincerely thank my incredible wife Deidre for being an amazing partner through everything in life. I appreciate all she has done to help me through the research and writing of this dissertation, both with love and support, in addition to patient proofreading - I am sure she has learned more about fault tolerance than she ever wanted to know. Truly, I would not have been able to accomplish this without her.

Contents

List of Figures

Chapter 1

Introduction

In recent decades, microprocessor performance has been increasing exponentially, due in large part to smaller and faster transistors enabled by improved fabrication technology. While such transistors yield performance enhancements, their lower threshold voltages and tighter noise margins make them less reliable [4, 36, 59, 61], rendering processors that use them more susceptible to *transient faults*. Transient faults are intermittent upsets caused by external events, such as energetic particles striking the chip, or internal events, such as noise couplings. These faults do not cause permanent damage, but may result in incorrect program execution by altering signal transfers or stored values. If this type of fault ultimately affects program execution, it is considered a *soft error*. Soft errors caused by transient faults have already caused significant failures in deployed systems. In 2000, Sun Microsystems acknowledged that cosmic rays interfered with cache memories and crashed server systems at major customer sites, including America Online, eBay, and dozens of others [5]. More recently, Hewlett Packard stated that a 2048-CPU server system in Los Alamos National Laboratory was frequently crashing because of cosmic ray strikes causing transient faults [29].

While transient faults are a concern for current processor designs, they will become a greater problem for future generations of processors. The reduction in transistor size is enabling the production of denser processor chips, and this higher density increases the probability that a transient fault striking the chip will affect a transistor. The lower threshold voltages of future processors are also reducing reliability as they increase the probability that an induced charge from an external event will change the transistor value. Due to the exponential increase in vulnerability, processor manufacturers are beginning to include reliability as a first order concern on par with performance, power, and cost.

1.1 Limitations of Existing Error Protection Techniques

To protect against transient faults, designers typically introduce redundant hardware. Storage structures such as caches and memory typically include extra information in the form of parity or error-correcting codes (ECC), which allow these hardware structures to detect and/or recover from such faults. However, applying such fine-grained protection to all hardware structures is prohibitively expensive. Bit-level techniques like ECC or parity are the traditional method of protecting value changes due to faults, but do not scale to protect all bits in the processor. They work well for infrequently changing storage structures, such as caches and memories, but cannot efficiently protect smaller collections of bits or those that are more frequently accessed. For example, protecting the register file with ECC is extremely costly in terms of both performance [68] and power [45]. Also, these bit-level techniques are problematic when the data are used in combination logic, such as being computed via an Arithmetic Logic Unit (ALU), as they cannot be used without paying a significant penalty in area, power, and/or performance.

Processor manufactures have begun to add protection via redundancy at higher levels because the fine-grained, bit-level protection techniques are unable to scale. These techniques, which are classified as *macro-reliability* protection, duplicate coarse-grained structures such as processor cores [22, 63, 79] or hardware contexts [57, 56, 18, 32, 50, 71] in order to provide transient fault tolerance in a manner that is more cost-effective and scalable than bit-level techniques. Macro-reliability techniques duplicate these larger structures and validate the data only when they propagate out of the structure. Delaying the comparison of data until they leave the redundant structure makes the fault validation more efficient. Compared with bit-level techniques which need to validate the data at each computation, macro-reliability techniques conduct comparisons less frequently and on reduced data sets. These techniques tend to be simpler to manufacture and have better performance because the duplication and validation are done off the critical path. Macro-reliability techniques are currently implemented in reliability-focused systems like the Compaq NonStop Himalaya [22], IBM z900 [63], and the Boeing 777 [79].

There are currently no existing techniques, either at the bit-level or macro-redundancy, that are able to provide scalable and efficient protection. Bit-level techniques are able to provide precise protection to those processor aspects which are most vulnerable, but are not able to scale to protect larger regions of the chip. Macro-reliability techniques are able to scale to provide protection for the entire processor, but they provide the same uniform protection regardless of the actual vulnerability and reliability requirements. Existing macro-redundancy techniques do not consider the native reliability of the system nor do they consider the tradeoff between performance and reliability, which may vary significantly for different regions. Regardless of the performance degradation or the reliability benefit, these implementations uniformly apply the same reliability technique to the entire system. This excessive overprotection is inefficient as it degrades performance, consumes power, and increases manufacturing costs.

1.2 Research Objectives and Contributions

To overcome the limitations of uniform protection in a scalable way, this dissertation introduces software modulation of the reliability level as a means to efficiently protect against transient faults. This dissertation also introduces reliability techniques that offer a spectrum of tradeoffs and, in addition to providing increased reliability, are easily configurable and offer efficient protection via software modulation.

This dissertation first presents four macro-redundancy techniques that provide a range of tradeoffs with respect to protection, performance, and hardware. While these techniques by themselves are alternatives to hardware-only reliability techniques, this dissertation will later explicate how they enable efficient configurable protection.

This dissertation presents a new software-only fault protection implementation, SWIFT (*Software Implemented Fault Tolerance*), which increases reliability by altering the application compilation process to insert redundant sequences of computation. It also includes validation code before all critical instruction to detect transient faults before they are propagated to externally visible data. SWIFT only implements fault detection; upon failed data validation, the application will exit and notify the user that it has found an error. Some users may prefer to have the application recover and continue to make forward progress in the presence of faults. For these situations, an extension of the SWIFT implementation, SWIFTR (*Software Implemented Fault Tolerance with Recovery*), can be used. SWIFTR includes the original and two additional and independent sequences of computations so that upon error detection, it can perform forward error recovery.

Although software-only techniques are advantageous because they have fewer requirements and lower costs, they may not provide adequate protection in all situations. For systems where software-only reliability is insufficient but hardware-only reliability is too costly, there is currently no solution for designers. A hybrid technique that combines both software and hardware aspects may be best able to satisfy the performance, reliability, and hardware constraints. This dissertation proposes two hybrid protection techniques, a detection-only technique, CRAFT (*Compiler Assisted Fault Tolerance*), and a detection-and-recovery technique, CRAFTR (*Compiler Assisted Fault Tolerance with Recovery*). These implementations augment the software-only SWIFT and SWIFTR techniques with structures inspired by the hardware-only Redundant Multithreading technique [32, 50]. The

CRAFT/CRAFTR techniques provide improved performance and reliability compared to software-only techniques, but require a processor designed with minimal specific hardware structures. The CRAFT and CRAFTR implementations are the first full processor hybrid techniques and provide an alternative to the two available design space extremes: they offer better reliability and performance than software-only techniques with lower design and manufacturing costs than hardware-only techniques. The SWIFT, SWIFTR, CRAFT, and CRAFTR macro-redundancy techniques are scalable like other macro-redundancy techniques but suffer from the same inefficiencies of uniform protection. However, these techniques are advantageous because they can be precisely configured to provide more efficient protection.

This dissertation proposes Software Modulated Fault Tolerance (SMFT) to maximize fault coverage while minimizing the performance impact of reliability techniques. This dissertation uses SMFT to take advantage of two key insights. First, not all faults in an unprotected application will cause an incorrect answer. For example, a fault to a register value outside of its live range will not change the program execution because that value will be overwritten before it is used. Due to this and other masking factors, unprotected applications have a natural degree of fault resistance. The natural resilience varies not only from application to application, but also within an application among regions and registers. SMFT extracts this information and uses it to reduce the added protection for regions that are already naturally resilient. Second, not all parts of the program will respond in the same way to reliability techniques. Regions may vary in protection costs and degree of benefit derived from added redundancy. SMFT identifies those regions which are most susceptible to faults as well as those which are inexpensive to protect. SMFT uses information about the effusiveness of the reliability technique along with the vulnerability information for each region to selectively apply the redundancy transformation. Thus, SMFT allows designers to trade off at a fine granularity performance for reliability and vice versa to protect the application in the most efficient manner.

1.3 Dissertation Organization

Chapter 2 provides background information on the current state of transient faults in processors. Chapters 3 and 4 explain the proposed software-only and hybrid fault protection techniques, respectively. These techniques provide protection as macro-redundancy techniques and serve as the baseline for configurable protection. Chapter 5 provides details on the methodology used to evaluate the techniques and configurations presented in this dissertation. The implementations are evaluated in term of reliability, performance, and hardware. Chapter 6 evaluates the software and hybrid techniques using this methodology. Chapter 7 shows the inefficiencies of existing full protection techniques and explores the benefits of configured protection. A quantitative evaluation of the configurations via an exploration of three specific constraints is given in Chapter 8. Finally, Chapter 9 discusses future avenues of research and summarizes the conclusions of this dissertation.

Chapter 2

Transient Fault Primer

This chapter provides an introduction to the nature of transient faults. Section 2.1 defines the reliability problem and explains how transient faults cause soft errors. Section 2.2 discusses the scope of the problem in current and future generations of processors. Current methods for mitigating transient faults and the limitations of these approaches are evaluated in Section 2.3.

2.1 Transient Fault Definition

A transient fault is an intermittent upset caused by external events, such as an energetic particle interfering with the processor. Internal events, such as noise coupling, can also have an effect although faults due to these events are less common. Both events can cause additional charge to be deposited and alter the value of a transistor. Despite this interference, the transistor will not sustain permanent damage and will respond normally upon the next active use. If a fault of this type ultimately affects program execution, it is considered a *soft error*.

The energetic particles that cause transient faults can be high energy neutron from cosmic radiation or alpha particles from contaminated packaging. Transient faults due to alpha particles were the predominant source of soft errors when they were first noticed in the late

1970's. Intel [66] and IBM [80] noticed that radioactive contaminants in the packaging of chips caused changes in the stored state of the processor. Manufacturing processes have since been updated to prevent such issues, yet cosmic radiation still plays a role in causing transient faults.

The effects of cosmic radiation on current processors are significant but can be hard to define because they are subject to changes in altitude, geography, and periodic solar phases. Depending on solar cycles, the amount of terrestrial neutrons (particles that penetrate the Earth's atmosphere and magnetic field) can vary by $\pm$ 20%. The neutron rate fluctuates on an 11-year solar cycle, with 2007 being the minimum for this current cycle [81]. Geography can also play a significant role in the rate of transient faults. A majority of solar radiation is deflected from the surface of the planet by Earth's magnetic field, but the intensity of the field is not uniform across all latitudes and longitudes. Depending on the geographic location, there can be a 2x difference in the amount of state-affecting radiation [81]. The altitude of the processor also has a significant effect on the transient fault rate. Using New York City at sea level as a baseline, the radiation increases 4x when moving up to Denver, Colorado and increases to 13x at Leadville, Colorado (the highest city in the United States). At an altitude of airplane travel the transient fault rate increases even further to 300x. Decreasing altitude below sea level has the inverse effect. For example, moving 20 meters underground reduces the cosmic radiation to 3% of sea level [1]. Although underground computing facilities can be used to shield cosmic radiation and limit transient faults, this strategy is too expensive and impractical for wide-spread use. These variations in cosmic radiation can noticeably change the processor vulnerability and lead to incorrect reliability estimates.

2.2 Scope of Transient Fault Problem

Transient faults leading to soft errors have already caused significant failures in deployed systems. In 2000, Sun Microsystems acknowledged that cosmic rays interfered with cache memories and crashed server systems at major customer sites, including America Online, eBay, and dozens of others [5]. More recently, in 2005, Hewlett Packard stated that a 2048-CPU server system in Los Alamos National Laboratory was frequently crashing because of cosmic ray strikes causing transient faults [29]. Cypress Semiconductor has publicly stated the

> *"wake-up call came in the end of 2001 with a major customer reporting havoc at a large telephone company. Technically, it was found that a single soft fail (a spontaneous bit-flip without a hardware defect) was causing an entire interleaved system farm (hundreds of computers) to crash. Another incident occurred at an automotive supplier, where their billion-dollar factory ground to a halt every month due to what was traced to a single bit-flip in their network."* [81]

In addition to causing machine crashes, recent research has shown that soft failures can lead to the derivation of secret keys in RSA public-key cryptography [3, 9] and induce security vulnerabilities in the Java Virtual Machine [20].

Future generations of processors will be even more susceptible to transient faults. Competing factors for the reliability of individual bits in a processors suggest the failure rate per bit will remain constant for the next few generations [21, 23]. The reduction in feature size for each transistor decreases the probability that cosmic radiation will strike the transistor, but the transistor also has a reduced critical voltage (the charge necessary to change its values), increasing the likelihood that a particle strike will affect the stored state. While the failure rate per bit is staying roughly constant, the failure rate per processor is increasing proportional to the number of transistors on a chip. The number of transistors on a processor is increasing at an exponential rate, and this is the driving component of the increased concern for transient faults.

Although the actual failure rates of commodity processors are strictly held trade secrets, it is possible to speculate on these rates by combining public data from a variety of manufacturing companies. In 2003, Fujitsu released its fifth generation SPARC64 with 80% of its 200,000 latches covered by some form of error protection [2]. Also, VLSI publications have shown average failure rates per bit to be 0.01 - 0.001 $\frac{FIT}{bit}$ [21, 23, 35, 67]. FIT (Failures in Time) is a commonly used metric for failure rates. It defines the number of failures for a given bit, structure, or processor, in one billion (10^9) hours. Chapter 5 describes FIT in detail along with other corresponding failure metrics. By using this SPARC64 processor as a baseline along with a conservative failure rate of 0.01 $\frac{FIT}{bit}$ and the historic knowledge that each successive processor generation doubles the number of latches roughly every 18 months, it is possible to estimate the failure rate of future generations of processors. These estimates only represent the latches and state elements of the processor. Although combinational circuits are not as susceptible to transient faults now, researchers predicts within the next decade the error rate of combinational logic will be comparable to that of state elements [61]. Assuming a 0.01 $\frac{FIT}{bit}$ rate, the failure rate of the SPARC64 processor was 2000 FIT in 2003. Although not a current industry estimate, a typically assumed failure rate target is 114 FIT (which corresponds to approximately 1000 years between failures) [11]. The SPARC64 processor protected 80% of its latches and thus left 20% vulnerable. Its 80% protection reduces the failure rate from 2000 to 400 FIT. If the less conservative 0.001 $\frac{FIT}{bit}$ estimate is used, the SPARC64 error rate is revised to 40 FIT, which is within the 114 FIT target.

Just as a transient fault may strike a processor but not cause a bit flip, a bit flip may not cause a noticeable program deviation. The ratio of bits that are required for program correctness compared to all of the bits in the system is known as the Architectural Vulnerability Factor (AVF) [76]. A reduction in AVF results in a decrease in the overall vulnerability of the system. The AVF of the processor is a combination of the AVF of the comprising structures, which range from 0-100% [6, 24, 76]. Chapter 5 covers AVF in greater detail, as AVF is one of the metrics used to compare the reliability techniques in this dissertation.

The AVF should be used in combination with the number of transistors and failure rate per bit when estimating the overall processor failure rate. The SPARC64 processor in 2003 was within the 114 FIT target if a $\frac{FIT}{bit}$ of 0.001 is assumed. Using the more conservative 0.01 $\frac{FIT}{bit}$, the processor needs an overall AVF of 28.5% or less to achieve the failure target. To achieve the reliability target with a processor in 2008 (with an estimate FIT of 20,000) the processor would need a combined AVF of 2.7% or less assuming 80% of its latches are protected by bit-level techniques like the SPARC64. Figure 2.1 shows failure rates for different AVF and bit-level protection percentages assuming a $\frac{FIT}{bit}$ of 0.001. The SDC FIT rate is plotted logarithmically for different years, starting in 2003. The horizontal solid line corresponds to the target FIT of 114. The upper dashed line represents the failure rate assuming no latches are protected by bit-level techniques and the AVF of the processor is 100%. With these assumptions, processors after 2003 are unable to meet the 114 FIT target. The lowest line corresponds to processors with 80% of the latches protected by bit-level techniques and an AVF of 30%. Even with this estimate, 2008 is the last year that this protection will be sufficient to meet the target FIT. Overall system vulnerability is increasing at an exponential rate, but it is becoming more difficult to increase the percentage of latches protected with traditional bit-level techniques. To maintain acceptable levels of reliability, manufacturers are moving to reliability methods that reduce the overall processor AVF.

2.3 Range of Methods for Fault Mediation

Reliability is becoming a first order design concern along with performance and power. When incorporating reliability in design decisions, computer architects have to manage the corresponding costs, whether in terms of design time, physical area, performance reduction,

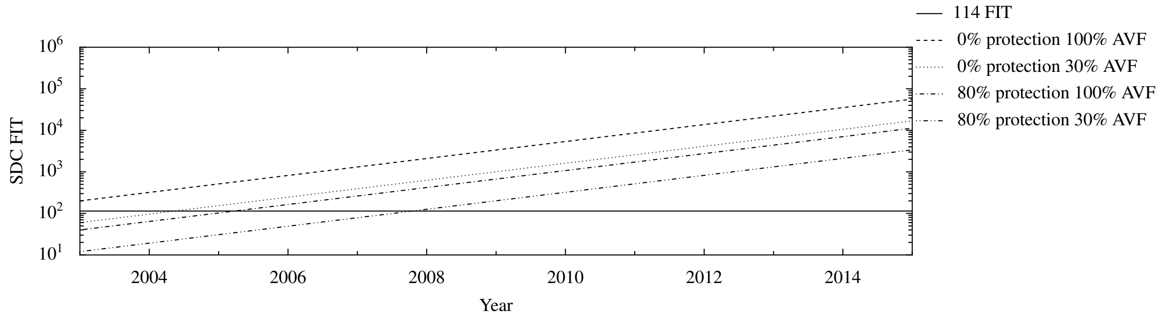

Figure 2.1: Estimated Failure Rates for Various AVF and Bit-Level Protection Percentages Assuming 0.001 FIT/bit

power increase, or end user costs. Generally, the goal is to reduce the failure rate to below an acceptable threshold, or in some cases to minimize the failure rate as much as possible while maintaining the reliability budget. Although the threshold or achieved failure rate may be very low, it is far too expensive (and often times impossible) to achieve a zero failure rate.

Traditional failure mitigation techniques have focused on low-level implementations. These techniques, including bit parity and ECC, are commonly used to protect state elements from single faults. These techniques increase reliability by adding redundancy in the form of extra bits which contain information that can detect and often correct a single bit change. These types of low level techniques are problematic when data are used in combination logic, such as being computed via an ALU. Bit-level protection techniques cannot be used for combinational logic without paying a significant penalty in area, power, and/or performance. Such techniques work well for infrequently changing storage structures such as caches and memories but do not work well for smaller collections of bits or those that are more frequently accessed. For example, protecting the register file with ECC has been shown to be extremely costly in terms of both performance [68] and power [45]. The most recent 1.72 billion transistor, dual-core Itanium processor is able to use parity detection on its floating-point and integer register file with minimal windows of vulnerability for approximately 10% additional area overhead [15, 27].

Bit-level techniques are unable to provide protection for the entire processor. Hence, computer architects have begun to implement reliability measures at higher levels. These macro-reliability techniques duplicate larger structures such as processor cores [22, 63, 79] or hardware contexts [18, 32, 50, 56, 57, 71] in order to provide transient fault tolerance in a manner that is more cost-effective and scalable. Such techniques duplicate larger structures and only validate the data when they propagate out of the structure. Delaying the comparison of data until they leave the redundant structure makes the fault validation more efficient as the comparisons occur less frequently and on a reduced data set. These macro techniques tend to be simpler to manufacture and have better performance because the duplication and validation is done off the critical path. The Compaq NonStop Himalaya [22], IBM z900 [63], and Boeing 777 [79] are all examples of high-reliability enterprise systems that employ macro-reliability techniques. While the techniques currently deployed may be effective, more efficient implementations that use smaller hardware changes and take advantage of time redundancy have been proposed. Proposals have been published for hardware-only [18, 32, 50, 71] and software-only [13, 39] macro-reliability techniques. The software-only SWIFT/SWIFTR techniques and the hybrid software-hardware CRAFT/CRAFTR techniques proposed in this dissertation are also macro-redundancy techniques with the added benefit that they can be precisely configured for efficient protection.

2.4 Limitations of Traditional Methods

Bit-level techniques are advantageous because they are able to provide precise protection to the processor aspects that are most vulnerable to faults, but they cannot scale to protect larger regions of the processor. Macro-reliability techniques are scalable but inefficient, as they provide uniform protection to the entire processor regardless of the actual vulnerability and reliability requirements. In addition, existing macro-redundancy techniques do not consider the tradeoff between performance and reliability. Both the performance cost and reliability increase attributed to a redundancy implementation can vary significantly for different regions. While the transformation may be appropriate for some regions, it unnecessarily increases the reliability for others. This overprotection degrades performance, consumes power, and increases manufacturing costs beyond what is required. To overcome the limitations explained above, this dissertation introduces software modulation of the reliability level as a means to efficiently protect against transient faults.

Chapter 3

Software-Only Fault Tolerance

This chapter describes two proposed software-only fault-tolerance techniques that provide overall processor core protection with minimal execution time overhead. These techniques do not require any hardware for execution and enable application protection by building redundancy directly into the compiled code. This chapter explains the benefits of these software-only techniques compared with other software and hardware techniques in Section 3.1. It also details the overall reliability strategy of the two techniques, first focusing on detection in Section 3.2 and then recovery in Section 3.3. It also covers the inherent vulnerabilities of software-only techniques in Section 3.4.

3.1 Comparison to Alternative Techniques

Software-only approaches to fault detection and recovery can significantly improve reliability and are advantageous because they do so without requiring any hardware modifications. Thus, software redundancy techniques are significantly cheaper, quicker, and easier to deploy. The process of transforming from a vulnerable system to a reliable system only requires recompilation of the running application. Hardware-only approaches, which comprise a majority of fault protection techniques, require the purchase and deployment of new machines.

Software-only techniques are also beneficial because they allow decisions of when and how to increase reliability to be made by the application developer, distributer, or end user. This is in contrast to hardware-only systems which require that reliability options be determined by the hardware designer or manufacturer. The lack of hardware requirements also allows software-only techniques to be applied to systems that are already deployed. Deployment of redundancy techniques to machines already in the field may become important due to poor estimates of the soft-error rate by designers or uncertainty regarding the usage conditions of the machine. Changes to the hardware operating environment can have noticeable effects on reliability, necessitating deployment of updated software redundancy techniques. Altitude, geography, and time can affect the error rate. For example, the soft-error rate from atmospheric neutrons is 4-5x higher in Denver than in New York City because of Denver's higher altitude [81].

The software-only techniques presented in this dissertation are the first to use a low level compilation transformation that significantly increases the protection of the processor core. Although processor caches are usually the most vulnerable aspect of the chip, those regions are almost always protected with some form of ECC. Previous techniques have implemented software-only protection using high-level code transformations in a pre-pass compiler [48] or by analyzing options for function or argument duplications [37]. Researchers have also proposed adding redundancy at the process [62], virtual machine [13], and hierarchy [77] of levels. For instance, Marathon recently began offering an enterprise system, everRun, that uses virtual machines to achieve reliability [14]. At the instruction level, researchers have also proposed modifying application data to hold augmented values that are resilient to single errors [39, 51]. Most closely related to this dissertation, researchers have proposed instruction duplicated techniques similar to SWIFT, but with a different model for memory correctness [40]. They do not synchronize at memory instructions, but instead duplicate the memory requirements.

```
add r1 = r2, r3
mul r1 = r1, 8
and r1 = r1, 0x11
st [r4] = r1
```

Figure 3.1: Bit Masking Code Example

3.2 Instruction Duplication and Validation

For the software-only techniques proposed in this dissertation, the redundant computation is inserted via a back end compilation phase. The SWIFT-enabled compiler replicates the original program's instructions and schedules the duplicates along with the original instructions in the same execution thread. This creates two independent versions of the computation in the application, therefore no single fault can affect both versions. The phase of the compiler that adds the reliable code is executed after most optimizations have been performed but before register allocation and scheduling. While possible to transform the code via binary translation [52], adding the reliability during compilation is much more efficient. The original and duplicate versions of instructions are inserted using virtual registers and are register allocated so they do not interfere with each another. Before certain synchronization points in the application, validation code sequences are inserted by the compiler to ensure that the data values being dynamically produced by the original and redundant instructions agree with each other. If these independent values do not agree, then a fault has occurred. As program correctness is defined by the output of a program, if memory-mapped I/O is assumed, then a program has executed correctly if all stores in the program have executed correctly. Consequently, store instructions must be used as synchronization points for comparison.

Delaying comparison until store instructions, rather than after each computation, significantly reduces the number of validation points and has two benefits. The delay improves performance by introducing less synchronization and checking instructions. It also reduces the chance that a fault that ultimately will not change the output of a program will cause the reliable version to halt (known as false detection). Figure 3.1 shows a code example that

```
1 add r1 = r2, r3
2
3 mul r1 = r1, 8
4
5
6
7 st [r1] = r2
```

(a) Original Code

```
1 add r1 = r2, r3
2 add r1'= r2',r3'
3 mul r1 = r1, 8
4 mul r1'= r1',8
5 br faultDet, r1 != r1'
6 br faultDet, r2 != r2'
7 st [r1] = r2
```

(b) SWIFT Transformation

Figure 3.2: Software-Only Transformation for Duplication and Validation

contains bit masking which can lead to false detection for certain reliability techniques. A fault to register `r1` will only cause invalid data to be written to memory if the fault changes the least two significant bits. The other bits of the register are masked away by the `and` instruction (**3**). While a reliability transformation could validate after each instruction, this validation would significantly increase the runtime of the application and unnecessarily detect faults that would cause no program deviation, such as faults to the insignificant bits of register `r1`.

The techniques described in this dissertation can tolerate transient faults that occur in the processor core, including the processor's pipeline, functional units, register state, etc. A Sphere of Replication (SoR) [50] is used to delineate the region in which the technique tolerates faults. The SoR in this system is the boundary between the processor and memory, including the cache. The memory subsystem and caches are not protected by these techniques, as they can be effectively protected by ECC. Data passing inward through that boundary via a load are considered correct and need not be validated by the processor. The data are replicated upon loading and copied to the multiple instances of redundant execution. Data passing outward through the boundary via a store will be considered correct by memory and must be validated by the processor before leaving.

Figure 3.2 shows a sample code sequence before (a) and after (b) the SWIFT fault-detection transformation. The non-synchronizing instructions are duplicated in the reliable version but encoded to use the duplicated versions of the registers. The add instruction in the original code (**1**) is duplicated and inserted as instruction **2**. The duplicate instruction uses redundant versions of the values in registers `r2` and `r3`, denoted by `r2'` and `r3'`, respectively. The result is stored in `r1`'s redundant version, `r1'`. The same process is used for the multiply instruction. The original instruction (**3**) is duplicated and inserted using redundant registers as instruction **4**.

The values of `r1` and `r2` are used as source operands in the store instruction (**7**) in the original code. To avoid storing incorrect values in memory as well as storing values to incorrect addresses, both `r1` (the address) and `r2` (the data) must be validated to match their redundant copies. If a difference is detected between the original and redundant versions, then a fault has occurred and the appropriate handling code is executed, whether that be exiting or restarting the program or simply notifying another process. In this section, I focus on detection options. Recovery options will be discussed in Section 3.3. Instructions **5** and **6** comprise the validation of the address and data, respectively. If the original versions of `r1` and `r2` match their redundant copies, then the fault-detecting branches (**5** and **6**) are not taken and the store may proceed as normal. The checking instructions (**5** and **6**) must be executed before the actual store instruction (**7**) as they ensure its correctness. The three instructions are executed like regular dependent instructions and are not executed atomically. This introduces a window of vulnerability in the reliability technique as a fault can occur to either of the source operands between the validation and the use. These and other windows of vulnerability will be discussed in Section 3.4 and eliminated through the hybrid proposal in Chapter 4.

Just as store instructions, which move data out through the SoR, are treated as synchronizing instructions, load instructions, which bring data in through the SoR, are also treated as synchronizing instructions. Data delivered to the processor core from memory are assumed to be correct as there is only one copy. The loaded data need to be replicated as they enter the system. Figure 3.3 shows how to handle loaded data entering the SoR.

The original, unprotected code is shown in Figure 3.3(a). The first reliable version shows a transformation that uses two load instructions. In this case, the loads are treated

```

ld r3 = [r4]

add r1 = r2, r3

```

(a) Original Code

```

ld r3  = [r4 ]
ld r3' = [r4']
add r1 = r2 ,r3
add r1'= r2',r3'
```

(b) Reliability Transformation with 2 Loads

```
br faultDet, r4 != r4'
ld r3 = [r4]
mov r3'= r3
add r1 = r2, r3
add r1'= r2',r3'
```

(c) Reliability Transformation with 1 Load

Figure 3.3: Reliability Transformation for Load Instructions

as non-synchronizing and are duplicated like an `add` or `mul` instruction. The redundant load will use the redundant `r4'` address and write to the redundant register `r3'`. The two load instructions are independent from the compiler's view although they will dynamically access the same memory location in the absence of faults. In the presence of a fault, the load instruction will retrieve an incorrect value. When the retrieved data are used in a computation that feeds a store instruction, the computed data will from each version not match and the fault will be detected.

This duplication of load instructions will not always provide correct behavior in the absence of faults. When executing an application that uses shared memory, the dual load solution can cause a consistent, false error detection state. The two load instructions do not execute atomically. During the time window between the first and the second loads, another process using the same shared memory may change the value in memory. Upon correct execution of the second load, the application will retrieve a different value. This will eventually be detected as a fault when the data propagate to a store instruction. It is possible that the application could always enter this inconsistent state when executing the two loads and thus never complete, which is obviously an unacceptable solution.

Even in a single-threaded system, there are certain loads that will result in the same inconsistent state as the shared memory example because they cannot be executed twice. Uncachable loads, such as loads from memory-mapped I/O devices, will provide different results to two successive loads from the same address. This is analogous to the shared memory situation that will cause fault detection although there are no faults. It is imperative that both the original and redundant versions receive the same data. The problems due to the lack of atomic loads can be eliminated with additional hardware that will be described in the hybrid fault tolerance techniques in Chapter 4. To handle these cases in a software-only implementation, all load instructions are treated as synchronizing instructions. Only one version executes, and is validated before execution. The loaded value is then copied within the processor core to the redundant versions. Figure 3.3(c) shows the correct software-only transformation for load instructions. This version is implemented in the SWIFT transformation. Instruction **1** is inserted to validate the input to the load instruction (`r4`). This validation instruction is analogous to the instructions **5** and **6** of Figure 3.2(b) which validate the store instruction.

The value loaded from memory in register `r3` must be provided to the redundant version in register `r3'`. In Figure 3.3(b), this is accomplished by the redundant load instruction. In the single load version, the data are simply copied from the original version to the redundant version. In this example, that is accomplished by the `mov` operation of instruction **3**. Similar to the synchronizing store instructions and corresponding validation, windows of vulnerability exist around load instructions. These will also be addressed in Section 3.4.

Correct execution of an application is defined as producing the correct output. Assuming that output is produced via memory-mapped I/O, obtaining correct output requires executing the correct store instructions. This necessitates that the correct data are delivered to the store instructions and the correct sequences of store instructions are executed. Because the original and redundant versions of the application's computations are scheduled together, there is only one path of execution for both versions. If the wrong execution path is taken, the wrong versions of both the original and redundant code will be executed. This will result in matching but incorrect data computations. Upon propagation of these data to memory, the two versions will be compared and the control-flow affecting fault will not be detected. For this reason, control-flow protection, in addition to store and load protection, is also necessary.

```
1 add r1 = r2, r3
2
3
4
5 br CB_2 r1 == r4
```

(a) Original Code

```
1 add r1 = r2, r3
2 add r1'= r2',r3'
3 br faultDet, r1 != r1'
4 br faultDet, r4 != r4'
5 br CB_2 r1 == r4
```

(b) SWIFT Control-Flow Transformation

Figure 3.4: Control-Flow Protection

```
br faultDet, r4 != r4'
ld r3 = [r4]
mov r3'= r3
add r1 = r2, r3
add r1'= r2',r3'
mul r1 = r1, 8
mul r1'= r1',8
div r2 = r2, 7
div r2'= r2',7
br faultDet, r1 != r1'
br faultDet, r2 != r2'
st [r1] = r2
```

(a) Alternating Original and Duplicate Versions

```
br faultDet, r4 != r4'
ld r3 = [r4]
add r1 = r2, r3
mul r1 = r1, 8
div r2 = r2, 7
mov r3'= r3
add r1'= r2',r3'
mul r1'= r1',8
div r2'= r2',7
br faultDet, r1 != r1'
br faultDet, r2 != r2'
st [r1] = r2
```

(b) Schedule Optimized for Performance

Figure 3.5: Other Options in Redundant Execution Scheduling

As with store instructions, branch instructions are treated as synchronizing and must have their the input registers validated. Figure 3.4 shows the software-only transformation for a branch instruction. Branch instruction **5** has two source registers, `r1` and `r4`, and both of these source operands must be validated. In the reliable version in Figure 3.4(b), validation instructions **3** and **4** are added before instruction **5** to verify the two source operands. This transformation detects control-flow faults that cause the application to deviate to the wrong branch direction. There are orthogonal software-only methods to this control-flow protection which can detect if the program randomly jumps to the wrong instruction, which can happen if the Program Counter is struck by a fault or if an invalid immediate field is used for a branch target. Various solutions have been presented in the literature [10, 38, 41, 53, 58, 70] and are largely independent of the reliability added here.

In the example code sequences presented here, an instruction is immediately followed by its duplicate. Because the original and redundant versions are largely independent of each other, an optimizing compiler (or dynamic hardware scheduler) is free to schedule the original and redundant instructions to use available Instruction Level Parallelism, thus minimizing the performance penalty of the reliability transformation. The main dependence restriction is the completion of the two versions of computation before fault validation. Other than the synchronizing instruction, the two code sequences have flexibility in execution ordering.

Figure 3.5 shows two sample instruction orderings for the reliable software-only transformation. Figure 3.5(a) shows the original and duplicate versions being executed in alternating sequence. Figure 3.5(b) shows an alternate execution sequence where, within the boundaries of the synchronizing instruction, the original version is executed fully before the redundant version. While this may be optimal from a performance standpoint, it creates a larger window of vulnerability between instructions **2** and **6**.

3.3 Software-Only Recovery Implementations

The software-only reliability transformations presented thus far have shown how to detect transient faults throughout the processor core. However, detecting faults is only part of the path to full fault tolerance. In order to truly be reliable, a system must also be able to recover from faults. Although detection prevents faults from corrupting visible data and enforces data integrity, it does not allow the application to correctly run to completion in the presence of a fault.

The SWIFT transformation can be seen as a double-modular redundancy implementation. In software, there are two copies of the data and instructions, one for the original and one for the redundant versions. This redundancy provides detection, but not recovery. In order to achieve recovery, it is natural to move to a triple-modular redundancy (TMR) implementation. This dissertation presents a software-only recovery transformation which does precisely that, SWIFTR (*Software Implemented Fault Tolerance with Recovery*). In-

```
1
2  ld r3  = [r4]
3
4
5  add r1 = r2, r3
6
7
8  mul r1 = r1, 8
9
10
11
12
13 st [r1] = r2
```

(a) Original Code

```
1  majority(r4,r4',r4'')
2  ld r3  = [r4]
3  mov r3' = r3
4  mov r3''= r3
5  add r1  = r2  , r3
6  add r1' = r2' , r3'
7  add r1''= r2'', r3''
8  mul r1  = r1  , 8
9  mul r1' = r1' , 8
10 mul r1''= r1'', 8
11 majority(r1,r1',r1'')
12 majority(r2,r2',r2'')
13 st [r1] = r2
```

(b) SWIFTR Transformation

Figure 3.6: Software-Only Recovery Transformation

stead of creating one redundant copy as in SWIFT, the SWIFTR transformation creates two redundant copies. Three independent copies allow a fault to corrupt any one version's computation while the two other versions can still have the correct computation. By using a simple majority voting scheme, any single-bit fault can be corrected as two of the versions will still maintain the correct data.

The code example from Figure 3.2 is shown again in Figure 3.6, but the SWIFTR recovery transformation is shown instead of SWIFT detection transformation. Instruction **6** of Figure 3.6 duplicates the previous `add` instruction, just as in SWIFT. The SWIFTR transformation, however, also inserts instruction **7**, a third version of the `add` instruction which uses a third set of registers, here denoted by `r1''`, `r2''`, and `r3''`. The `mul` instruction (**8**) is similarly replicated two additional times with redundant registers as instructions **9** and **10**. For the same uncachable and multi-access memory reasons discussed in Section 3.2, after the `ld` instruction (**2**), the data are copied into the redundant versions. In addition to the single move instruction for detection (**3**), the SWIFTR transformation also inserts a second move instruction (**4**) to enable recovery.

Furthermore, the fault detection code used by SWIFT has been replaced, in SWIFTR, by recovery code at instructions **1**, **11**, and **12**. The recovery code is simply a majority voting procedure. If two versions of a register, `r1` and `r1'` for example, have the same value, but the third version, `r1''`, does not, then `r1''` is set to the value in `r1` or `r1'`. This assignment corrects the corrupted value of `r1''`. The majority correction is required in the same situations that require detection instruction in SWIFT, before any synchronizing instructions.

The SWIFTR transformation uses TMR for its recovery and is considered a forward error recovery technique because errors are correct as execution progresses. Alternatives to this method include backward error recovery techniques which are usually done via checkpointing and rollback mechanisms. In backward error recovery systems, certain parts of the processor state are stored away during the normal execution of the program. In the event of a fault detection, the stored data are retrieved and the current data are overwritten so the program can re-execute from the recovery point. Essentially, checkpointing allows the application to move backwards in time to re-execute part of the instruction stream without side effects. The fault model in this dissertation is transient, thus re-execution of the instructions will not re-introduce errors due to the delay in time between the first and second attempts. Backwards error recovery systems using checkpointing have been proposed and implemented in several instances [17, 46, 64, 65].

The rollback mechanism requires that no data leave the system until all data in the system can be verified. If some data exited the system's SoR, then they could not be discarded upon rollback, resulting in incorrect re-execution. Although the detection and re-steering of the execution path to recovery code can be implemented in software, the buffering of the hardware state until checkpointing would require the addition of specialized hardware. Previous approaches to checkpointing and recovery take advantage of additional hardware, but in that the additional hardware is unacceptable in software-only techniques (and too costly for a technique focused on minimal hardware), the inline recovery method is preferred.

3.4 Undetected Errors

Although the protection offered by these software-only techniques is quite robust, there are five fundamental limitations to all software-only methods:

(1) *Between validation and use* – As redundancy is introduced solely via software instructions, any time a fault affects the value used by the store but not the corresponding value used by the compare, an error may go undetected. For simple microarchitectures, this is equivalent to saying that an error may go undetected if a fault occurs after the validation instruction but before the store instruction. In more complicated microarchitectures, errors to a register may manifest themselves on some instructions and not others due to the modern design of instruction windows and bypass networks. In these situations, it is more difficult to enumerate the conditions under which a store can be affected but not its corresponding validation instructions.

(2) *Certain opcode faults* – It is possible for an opcode fault to change a regular instruction into a synchronizing instruction. For example, a strike to an instruction's opcode bits may change a non-store instruction into a store instruction. As store instructions are not duplicated by the compiler, there is no way of detecting and recovering from this situation. The incorrect store instruction will be free to execute and the value stored will corrupt memory which may result in silent data corruption. A non-branch instruction may be converted to a branch instruction by a single bit flip, in which case the program is susceptible to the control-flow issues discussed below.

(3) *Microarchitectural state* – Because the microarchitectural state of the processor is not exposed to the application, software-only techniques are unable to protect against faults on some portions of that state, such as parts of the control logic. If faults to such state cause deadlock, any software-only scheme will be unable to make forward progress. For example, if a fault causes an instruction that has become ready to execute to be marked as stalled, the instruction may never be execute and the application

may never finish [7].

(4) *Control-flow errors* – If a control-flow error occurs such that the program branches to a block and begins executing from the same point in the original and redundant computation streams, or directly to a store, system call, or other potentially disastrous instruction, then a potentially erroneous value may propagate to memory. This effect can be minimized by separating the two computation streams as much as possible and by performing validation checks as close as possible to every potentially output-corrupting instruction. This will reduce the windows of vulnerability (although not eliminate them) and increase the performance penalty. Although additional software-only control-flow redundancy can reduce control-flow errors, these errors cannot be completely eliminated.

(5) *Single load value* – Instead of duplicating load instructions, SWIFT simply adds a move instruction to copy the loaded value into a redundant register, as in instruction **2** of Figure 3.3. Before the copy is performed, the program only has one version of the loaded value. If a fault occurs to this value before it is copied, then the incorrect value will be propagated to the redundant register and the error will go undetected.

The subsequent chapter of this dissertation will show how the addition of two minor hardware structures can be used to reduce these fundamental limitations of software-only techniques.

Chapter 4

Hybrid Fault Tolerance

This chapter presents alternatives to the software-only techniques presented in the previous chapter. The techniques in this chapter use a combination of software-only and hardware implementations to provide a hybrid approach to fault tolerance that offers better performance and higher reliability than the software-only approaches, but is still configurable at a fine granularity.

The two low-cost hybrid hardware/software redundancy techniques presented in this chapter are CRAFT (*CompileR-Assisted Fault Tolerance*) and CRAFTR (*CompileR-Assisted Fault Tolerance with Recovery*). They are modeled after the SWIFT and SWIFTR implementations but have minimal hardware additions to increase performance and remove certain vulnerabilities. The CRAFT/CRAFTR techniques introduce hardware that is inspired by the hardware-only technique of Redundant Multithreading (RMT) [32, 50].

4.1 Removing the Store Vulnerability

As noted in Section 3.4, software-only techniques contain store and other synchronizing instruction that become single points of failure and make the application vulnerable to strikes during the time interval between the validation and the use of a register values. For example, consider the code snippet given in Figure 4.1. Figure 4.1(a) shows the orig-

```
1 add r1 = r2, r3
2
3 mul r1 = r1, 8
4
5
6
7 st [r1] = r2
```

(a) Original Code

```
1 add r1 = r2, r3
2 add r1'= r2',r3'
3 mul r1 = r1, 8
4 mul r1'= r1',8
5 br faultDet, r1 != r1'
6 br faultDet, r2 != r2'
7 st [r1] = r2
```

(b) SWIFT Transformation

```
1 add r1 = r2, r3
2 add r1'= r2',r3'
3 mul r1 = r1, 8
4 mul r1'= r1',8
5
6 st'[r1']= r2'
7 st [r1] = r2
```

(c) CRAFT Transformation

Figure 4.1: CRAFT Store Transformation

inal, unprotected version of a small code segment. Figure 4.1(b) shows the code after the software-only reliability transformation. If `r1` receives a strike after instruction **5** is executed but before instruction **7** is executed, then the value will be stored to the incorrect address. Similarly, if a fault occurs on `r2` after instruction **6** but before instruction **7** then an incorrect value will be stored to memory.

In order to protect data going to memory in the CRAFT technique, the compiler duplicates store instructions in the same way that it duplicates all other instructions, except that store instructions are also tagged with a single-bit version name, indicating whether a store is an original or a duplicate. Figure 4.1(c) shows the result of this transformation. The original store (**7**) is duplicated as instruction **6**. This removes the window of vulnerability of the software-only store instructions. This also allows the hybrid technique to more freely schedule the two versions of the computation because synchronizing instructions **5** and **6** of Figure 4.1(b) are no longer necessary. This will be discussed further in Section 4.4 and evaluated in Section 6.1.

The modified hybrid code is run on hardware with an augmented store buffer called the *Checking Store Buffer* (CSB). The CSB functions much as a normal store buffer, except that it does not commit entries to memory until they are validated. An entry becomes validated once the original and the duplicate version of the store have been sent to the store buffer, and the addresses and values of the two stores match perfectly. The CSB can be implemented by augmenting the normal store buffer with a tail pointer for each version and a `validated` bit for each buffer entry. An arriving store first reads the entry pointed to by the corresponding tail pointer. If the entry is empty, then the store is written in it, and its `validated` bit is set to false. If the entry is not empty, then it is assumed that the store occupying it is the corresponding store from the other version. In this case, the addresses and values of the two stores are validated using simple comparators built into each buffer slot. If a mismatch occurs, then a fault is signaled. If the two stores match, the incoming store is discarded, and the `validated` bit of the already present store is turned on. The appropriate tail pointer is incremented modulo the store buffer size on every arriving store instruction. When a store reaches the head of the store buffer, it is allowed to write to the memory subsystem if and only if its `validated` bit is set. Otherwise, the store stays in the store buffer until a fault is raised or the corresponding store from the other version enters the buffer.

The store buffer is considered clogged if it is full and the validated bit at the head of the store buffer is unset. Both tail pointers must be checked when determining whether the store buffer is full, as either version 1 or version 2 stores may be the first to appear at the store buffer. A buffer clogged condition may occur because of faults resulting in bad control flow, version bit-flips, or opcode bit-flips, all of which result in differences in the stream of stores from the two versions. If such a condition is detected at any point, then a fault is signaled. To prevent spurious fault signaling, the compiler must ensure that the difference between the number of version 1 stores and version 2 stores at any location in the code does not exceed the size of the store buffer.

The checking store buffer also eliminates many opcode and control-flow errors related to store instructions. As mentioned before, if an opcode is hit with a fault and changes an instruction into a store instruction, the software-only version could not detect that. Similarly, if a control-flow fault causes the program to start executing at a store instruction, the software-only implementation could not recover. As all memory data are sent through the CSB, the invalid data in those cases would be held in the CSB. Those cases would not corrupt external memory because the redundant store that validates the data would never

```
1
2 ld r3 = [r4]
3
4 add r1 = r2, r3
5
6 mul r1 = r1, 8
7
```

(a) Original Code

```
1 br faultDet, r4 != r4'
2 ld r3 = [r4]
3 mov r3'= r3
4 add r1 = r2, r3
5 add r1'= r2',r3'
6 mul r1 = r1, 8
7 mul r1'= r1',8
```

(b) SWIFT Transformation

```
1
2 ld r3 = [r4]
3 ld r3' = [r4']
4 add r1 = r2, r3
5 add r1'= r2',r3'
6 mul r1 = r1, 8
7 mul r1'= r1',8
```

(c) CRAFT Transformation

Figure 4.2: CRAFT Load Transformation

be executed.

4.2 Removing the Load Vulnerability

The previous chapter explained how executing two load instructions would not be a correct implementation for redundancy because that implementation could not guarantee forward progress of the application. The software-only implementations in this dissertation correctly implement load transformation by generating a compare instruction before and a move instruction after every load. Figure 4.2 shows an original code segment, as well as the software-only and hybrid reliability transformations. Figure 4.2(b) shows the validation before (**1**) and move after (**3**) the load instruction of the software-only version.

This produces two windows of vulnerability, namely vulnerabilities (**1**) and (**5**) in Section 3.4. In Figure 4.2(b), if a fault occurs in `r4` after instruction **1** but before instruction **2**, then the load instruction will load from an incorrect address. If a fault occurs in `r3` after instruction **2** but before instruction **3**, then a faulty value will be duplicated into both streams of computation and the fault will go undetected.

In order to remove these windows of vulnerability, the hybrid techniques must execute a redundant load instruction. An application of the hybrid transformation is shown in Figure 4.2(c). Instead of copying the value from a single load it would ideally execute a second, redundant load like instruction **3**. Unfortunately, merely duplicating load instructions will not provide correct redundant execution in practice. Treating both versions of a load as normal loads will lead to problems in multi-programmed environments as any intervening writes by another process to the same memory location can result in a false detection. In such cases, the technique prevents the program from running to completion even though no faults have been actually introduced into the program's execution.

The CRAFT and CRAFTR techniques make use of a protected hardware structure called the *Load Value Queue* (LVQ) to enable redundant load execution. The LVQ only accesses memory for the original load instruction and buffers the loaded value for the duplicate load. An LVQ entry is deallocated only if both original and duplicate versions of a load have executed successfully. A duplicate load can successfully receive its value from the LVQ only if its address matches that of the original load buffered in the LVQ. If the addresses mismatch, a fault has occurred and the appropriate action is taken. Loads from different versions may be scheduled independently, but they must maintain the same relative ordering across the two versions. Additionally, for out-of-order architectures, the hardware must ensure that loads and their duplicates both access the same entry in the LVQ.

The duplicated loads and the LVQ provide completely redundant load instruction execution. As the address validation is now done in hardware, the software address validation in instruction **1** of Figure 4.2 can now be removed. The LVQ also allows the compiler more freedom in scheduling instructions around loads just as the CSB allows the compiler more freedom in scheduling instructions around stores. This instruction optimization will be further discussed in Section 4.4. It is also important to notice that because the duplicate load instruction will always be served from the LVQ, it will never cause a cache miss or create additional memory bus traffic.

4.3 Hybrid Recovery

This section covers the hybrid recovery technique, CRAFTR. This technique is similar to the software-only recovery technique, SWIFTR, in its use of forward error recovery via

```
1
2 ld r3 = [r4]
3
4
5 add r1 = r2, r3
6
7
8 mul r1 = r1, 8
9
10
11
12
13 st [r1] = r2
```

(a) Original Code

```
1 majority(r4,r4',r4'')
2 ld r3  = [r4]
3 mov r3' = r3
4 mov r3''= r3
5 add r1  = r2  , r3
6 add r1' = r2' , r3'
7 add r1''= r2'', r3''
8 mul r1  = r1  , 8
9 mul r1' = r1' , 8
10 mul r1''= r1'', 8
11 majority(r1,r1',r1'')
12 majority(r2,r2',r2'')
13 st [r1] = r2
```

(b) SWIFTR Transformation

```
1
2 ld r3    = [r4  ]
3 ld r3'   = [r4' ]
4 ld r3''  = [r4'']
5 add r1  = r2  , r3
6 add r1' = r2' , r3'
7 add r1''= r2'', r3''
8 mul r1  = r1  , 8
9 mul r1' = r1' , 8
10 mul r1''= r1'', 8
11 st [r1' ] = r2'
12 st [r1''] = r2''
13 st [r1  ] = r2
```

(c) CRAFTR Transformation

Figure 4.3: Recovery Transformations

TMR. As CRAFT added hardware structures to eliminate certain windows of vulnerability in SWIFT, CRAFTR will use similar structures to remove the same windows of vulnerability in SWIFTR.

Figure 4.3 shows the application transformations from unprotected code sequence (a) to the recovery techniques in both SWIFTR (b) and CRAFTR (c). Just as CRAFT required the checking store buffer to detect faults, CRAFTR uses an augmented version of the checking store buffer to enable recovery. Instructions **11**, **12**, and **13** are all sent to the checking store buffer and the additional hardware will only send the address and data out to memory when it receives three store instructions. The CSB does a simple majority voting to correct any corrupted data before sending the store out of the SoR.

Also, like the CRAFT technique, the CRAFTR technique utilizes an LVQ to buffer the loaded value from the first store thus the second and third versions receive the same data. Instruction **2**, the first load, triggers the loaded value to be accessed from memory, and the subsequent loads, **3** and **4**, retrieve their data from the LVQ.

Both hybrid versions, CRAFT and CRAFTR, remove the windows of vulnerability from their respective software-only versions. However, because the software-only recovery technique has three rather than two versions, it has a greater number of windows of vulnerability. The CRAFTR technique removes five vulnerabilities, while the CRAFT technique

```
ld r3 = [r4]

add r1 = r2, r3

mul r1 = r1, 8



st [r1] = r2
```

(a) Original Code

```
br faultDet, r4 != r4'
ld r3 = [r4]
mov r3'= r3
add r1 = r2, r3
add r1'= r2',r3'
mul r1 = r1, 8
mul r1'= r1',8
br faultDet, r1 != r1'
br faultDet, r2 != r2'
st [r1] = r2
```

(b) SWIFT Transformation

```
ld r3 = [r4]
ld r3' = [r4']
add r1 = r2, r3
add r1'= r2',r3'
mul r1 = r1, 8
mul r1'= r1',8


st [r1] = r2
st'[r1']= r2'
```

(c) CRAFT Transformation

```
ld  r3  = [r4]
add r1  = r2, r3
mul r1  = r1, 8
st  [r1]= r2
ld  r3' = [r4']
add r1' = r2',r3'
mul r1' = r1',8
st'[r1']= r2'
```

(d) Alternate Schedule for CRAFT Transformation

Figure 4.4: Alternative CRAFT Transformation

removes three vulnerabilities. The five SWIFT vulnerabilities exist in Figure 4.3(b) between instructions **1-2**, **2-3**, **2-4**, **11-13**, **12-13**.

4.4 Comparison to Software-Only Techniques

The modest hardware additions allow the system to detect faults in store addresses and data, as well as dangerous opcode bit-flips, thus protecting against vulnerabilities (1) and (2) mentioned in Section 3.4. Although these techniques duplicate all stores, it is important to note that no extra memory traffic is created because there is only one memory transaction for each pair (or triplet) of stores in the code.

The memory load changes for the hybrid techniques, the duplicated loads and the LVQ, provide completely redundant load instruction execution. This protects against vulnerabilities (1) and (5) mentioned in Section 3.4. Also, because the duplicate load instructions will always be served from the LVQ, they will never cause a cache miss or use additional memory bandwidth.

In addition to the reliability benefits described in the previous sections, the additional hardware also provides optimization opportunities for the generated code. Figure 4.4 shows four versions of a code segment. Figures 4.4 (a), (b), and (c) show an abbreviated unprotected, software-only detection, and hybrid detection code examples, respectively. In Figure 4.4(c), the load address, store address, and store data validations are implemented in hardware so the synchronizing instructions required to validate the memory accesses are removed. The hybrid technique allows the elimination of instructions **1**, **8**, and **9** from the software-only transformation.

In the hybrid version, each version of the store instruction and the instructions they depend on, including any load instructions, can now be scheduled independently. Figure 4.4(d) shows an alternate scheduling for the hybrid technique that takes advantage of the removal of synchronizing instructions. This technique is able to schedule the instruction to minimize execution time without the burden of version synchronization. The performance and reliability benefits of these techniques will be quantitatively illustrated in Chapter 6. The net result is a system with enhanced reliability, higher performance, and only modest additional hardware costs.

The hardware structures that are added for this hybrid approach are very similar to the structures proposed for RMT [32, 50]. Unlike ECC, parity, or other bit-level techniques, the hardware structures that are necessary to implement the reliability technique, both for RMT and CRAFT, are outside of the core pipeline. This allows the processor design to be only slightly modified from the standard versions to add these structures.

Chapter 5

Evaluation Methodology

This chapter describes the various methodologies used to evaluate the reliability techniques proposed in this dissertation. The major tradeoffs between these techniques involve performance and reliability, and to a smaller extent hardware cost. The evaluation methodologies explained here will be used in the quantitative assessments in Chapter 6 and Chapter 8.

5.1 Performance Evaluation Methodology

All of the evaluated techniques are generated by a modified version of the VELOCITY [69] compiler for the IA-64 architecture targeting the Intel Itanium 2 processor. The compiler was modified to insert the additional reliability instructions when specified. The baseline programs are aggressively optimized VELOCITY compilations without any additional redundancy.

Performance was evaluated for a set of benchmarks drawn from SPEC 2006, SPEC 2000, SPEC 95, and MediaBench [25] suites. The performance evaluation was conducted by executing the resulting binaries on an HP workstation zx6000 with a 900Mhz Intel Itanium 2 processors running CentOS 4.4 with 4GB of memory. The `perfmon` [44] utility was used to measure the CPU cycles for each benchmark executed. The performance graphs throughout this dissertation are presented as Normalized Execution Time. The execution time of the reliable or partially reliable application is normalized to the execution time of the unprotected application.

5.2 Reliability Evaluation Methodology

This section describes the methodology used to evaluate all of the techniques presented in this dissertation. First, Section 5.2.1 reviews the standard transient fault model used in the dissertation. Section 5.2.2 explicates the two metrics used to compare reliability of the various techniques. Finally, Section 5.2.3 details the simulation and evaluation method used to determine the reliability metrics.

5.2.1 Fault Model

This dissertation proposes techniques to increase reliability in the presence of transient faults. Throughout this dissertation, the *Single Event Upset* (SEU) model is assumed. That is, the assumption is made that exactly one event occurs that changes exactly one bit during a program's execution. The value of the altered bit is always toggled to the opposite value. This upset model is the standard transient fault model used in the reliability literature [8, 31, 32, 37, 38, 40, 39, 47, 51, 53, 54, 55, 56, 60, 71].

5.2.2 Reliability Metrics

This section details the two metrics used to evaluate the reliability of the techniques. The first, Architectural Vulnerability Factor (AVF), captures the system reliability without focusing on the specific environmental and processes manufacturing factors. The second, Mean Work To Failure (MWTF), combines reliability and performance to measure how much work can be accomplished in the presence of failures.

Architectural Vulnerability Factor

FIT, which represents the number of failures for a given bit, structure, or processor, in one billion (10^9) hours and MTTF, the Mean Time to Failure, are commonly used metrics for overall system reliability. The FIT of a system is simply the inverse of the MTTF of that system. AVF is a metric recently used in reliability to separate inherent architectural reliability factors from environmental and manufacturing reliability factors.

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

FIT is a product of the AVF and raw error rate. The raw error rate is determined by the manufacturing processes and environmental factors. Faster clock rates, smaller transistors, and lower threshold voltages of new manufacturing processes can negatively affect the raw error rate. Environmental factors like altitude, shielding, and geography can also have a detrimental effect on the raw error rate. Using AVF as a reliability metric removes those factors from the reliability evaluation and focus on the architectural and microarchitectural decisions that affect reliability.

To compute the AVF, the system is analyzed to determine the properties of each bit at each moment in time. Any bit in the system at any given execution point can be classified as one of the following [33]:

ACE These bits are required for *Architecturally Correct Execution* (ACE). A transient fault affecting an ACE bit will cause the program to execute incorrectly.

unACE These bits are not required for ACE. A transient fault affecting an unACE bit will not affect the program's correct execution. For example, unACE bits occur in state elements that hold dynamically dead information, logically masked values, or control flows that are Y-branches [74]

The notion of an AVF can be defined as follows:

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

Transient faults in ACE bits can be further classified by how they manifest themselves in program output.

DUE A transient fault on an ACE bit that is caught by a fault detection mechanism is a *Detected Unrecoverable Error* (DUE). A detected error can only be considered DUE if it is fail-stop, that is, if the detection occurs before any errors propagate outside a boundary of protection. No fault can be a DUE in a non-fault-detecting system.

SDC A transient fault on an ACE bit that is *not* caught by a fault-detection mechanism will cause *Silent Data Corruption* (SDC). This could manifest itself as a spurious exception, an incorrect return code, or corrupted program output.

In recovery techniques, detected and recoverable faults are not considered errors. This dissertation will sometimes refer to a bit as being DUE or SDC. A DUE bit is an ACE bit which, if flipped by a transient fault, would result in a DUE. Similarly, an SDC bit is an ACE bit which, if flipped, would result in an SDC. Just as ACE bits were further categorized into DUE bits and SDC bits, AVF can be broken up into AVF$_{DUE}$ and AVF$_{SDC}$ by computing the ratio of DUE bits or SDC bits over the total bits in the structure, respectively.

The goal of any fault-detection system is to convert a system's AVF$_{SDC}$ into AVF$_{DUE}$. Fault-detection systems will have a higher AVF, the sum of AVF$_{SDC}$ and AVF$_{DUE}$, than the original system as will be shown quantitatively in Chapter 6. There are two principal reasons for this. First, most practical fault-detection schemes may exhibit false DUE, which arises whenever the system detects a fault in an unACE bit. Delaying validation until data leave the SoR minimizes the amount of false DUE, but the false detection still occurs because the system cannot determine whether a flipped bit is unACE, and thus may have to conservatively signal a fault. Second, any type of fault detection necessarily introduces redundancy, and this increases the number of bits present in the system. As all bits in the system are susceptible to transient faults, this also leads to a higher soft error rate. Although detection techniques often increase the overall AVF, they reduce AVF_{SDC}. SDC faults are more deleterious than DUE faults and system designers tolerate typically higher incidents of DUE in order to reduce SDC [11]. In recovery techniques, the SDC bits are converted directly into unACE bits and the overall AVF is lowered.

Mean Work To Failure

MTTF and AVF are two commonly used metrics to assess reliability, but may not be appropriate in all cases. This section presents MWTF, a metric that generalizes MTTF to make it applicable to a wider class of fault-detection systems and enables the evaluation of the techniques in this dissertation.

MTTF and FIT are generally accepted as appropriate metrics for system reliability. Unfortunately, these metrics do not capture the tradeoff between reliability and performance. For example, suppose that system A and system B were being compared, and that system A was twice as fast but half as reliable as system B. Specifically, let

$$\text{MTTF}_A = \frac{1}{2} \cdot \text{MTTF}_B \qquad\qquad t_A = \frac{1}{2} \cdot t_B$$

What is the probability of a fault occurring during the execution of the program is for each system? Although B has double the MTTF of A, the performance is such that any program must run for twice as long, and so the probability of failure *for the program* is equal for A and B. In this sense, they are equally reliable. However, the MTTF metric would counterintuitively select the slower of the two systems, B, as the most reliable.

In order address this issue when only comparing changes in microarchitectural decisions, Weaver et al. introduced the alternative *Mean Instructions To Failure* (MITF) metric [76]. This metric captures the tradeoff between performance and reliability for hardware fault-tolerance techniques – i.e. those which do not change the programs being executed, but which may affect Instructions Per Cycles (IPC) – but the metric is still inadequate for the general case where the program binary, and hence the number of instructions committed, can vary.

$$\text{MITF} = \frac{\text{number of committed instructions}}{\text{number of errors encountered}} \quad = \frac{\text{number of committed instructions}}{\frac{\text{execution time in cycles}}{\text{frequency} \times \text{MTTF}}}$$

$$= \text{IPC} \times \text{frequency} \times \text{MTTF} \qquad = \frac{\text{frequency}}{\text{raw error rate}} \times \frac{\text{IPC}}{\text{AVF}}$$

MITF has two components, as shown above. The frequency and error rate component represent the manufacturing and environmental factors of reliability. The IPC and AVF components represent the microarchitectural factors of reliability. When comparing machines that were built with the same manufacturing processes and are running in the same environment, the first component can be factored away. Only the microarchitectural factors are needed for comparison. In this case, MITF takes into consideration both the vulnerability and the execution time. However, this will only work when differences between the systems are limited to microarchitectural changes. If the systems require different executables, as is the case for software-only techniques, then MITF is no longer a valid metric for comparison.

To adequately describe the reliability for hardware *and* software fault-tolerance techniques, this dissertation introduces a generalization of MITF called MWTF:

$$\text{MWTF} = \frac{\text{amount of work completed}}{\text{number of errors encountered}} \quad = \frac{1}{\text{raw error rate}} \times \frac{1}{\text{AVF} \times \text{execution time}}$$

Similarly to MITF, MWTF can be abstracted into two components, one for manufacturing and environmental factors and another for architectural and microarchitectural factors. The execution time of MWTF corresponds to the time to complete one unit of work. A unit of work is a general concept whose specific definition depends on the system being evaluated. The unit of work should be chosen so that it is consistent across evaluations of the systems under consideration. For example, if one chooses a unit of work to be a single instruction, then the MWTF equation reduces to MITF. This is only appropriate for hardware fault-detection evaluation because the program binaries are fixed and an instruction represents a constant unit of work. In a server application it may be best to define a unit of work as a transaction. In other cases, work may be better defined as the execution of a program or a suite of benchmarks. For the evaluation of techniques in this dissertation, the unit of work used for MWTF is the execution of a program. With this definition of work, it is obvious that the previous example of halving the AVF, therefore doubling the MTTF, while doubling execution time will not increase MWTF. Regardless of the method (hardware or software) by which AVF or execution time is affected, MWTF accurately captures the reliability of the system. Such a metric is crucial when comparing hardware, software, and hybrid systems.

This dissertation evaluates the tradeoff between performance and reliability using normalized MWTF (nMWTF). This metric is a comparison of the MWTF of the reliability technique normalized to the MWTF of the baseline, unprotected technique. In this dissertation, it is assumed that the unprotected technique is manufactured using the same process technology as the reliability techniques and operated in the same environment. Like AVF, nMWTF is able to remove the environmental and manufacturing aspects of reliability to focus on the implementation's effects. By normalizing the reliability technique's MWTF to the unprotected MWTF, the process and environmental effects given by the raw error rate are canceled out and only the architectural factors remain. The nMWTF is the inverse of the product of the normalized AVF and normalized execution time:

$$\text{nMWTF} = \frac{\text{raw error rate} \times \text{AVF}_{base} \times \text{execution time}_{base}}{\text{raw error rate} \times \text{AVF}_{new} \times \text{execution time}_{new}}$$

$$= \frac{\text{AVF}_{base} \times \text{execution time}_{base}}{\text{AVF}_{new} \times \text{execution time}_{new}} \qquad = \frac{1}{\text{AVF}_{norm} \times \text{execution time}_{norm}}$$

5.2.3 Fault Injection

This section describes the simulation method used to determine the AVF, and thus MWTF, of the reliability techniques. Execution in the presence of transient faults is evaluated by simulating a fault injection into the running processor. Using the SEU model, exactly one bit is toggled in each execution. The application runs normally, and then a fault is simulated by toggling a bit in the application at a random point in time, uniformly distributed over the execution time of the program without faults inserted. This is implemented via a special signal handler to catch the timing signal. The application is paused, the chosen bit is toggled, and the application is allowed to continue executing. This is an advantageous way to conduct a simulation because the application can be run to completion. The result of the application clearly defines the reliability type of the bit: unACE or ACE. When a program output does not match exactly with the fault-free output, the bit that was changed is considered an ACE bit. If the output does not match the fault-free output because the application signaled a fault detection then the bit is a DUE bit; otherwise it is an SDC bit.

Incorrect execution is defined as producing an output that does not match the output in the absence of faults. When injecting faults, it is possible that the application may never finish if the toggled bit causes an infinite loop or invalid OS call. Also, the application may crash from application terminating signals such as a segmentation faults or divide by zero. These cases are considered to be incorrect executions and the bits affected are treated as SDC bits, although it is possible to argue that either case could be a detected fault if the application was known to not crash or known to take a certain maximum amount of time for the input.

Modeling via simulated fault injection is also advantageous because the simulation can be done quickly as the application can run at native speed. The overhead of the timing signal and fault injection are negligible. To get a statistically significant AVF distribution, many fault injections must be simulated. This can be done in parallel as each execution only has one fault and is independent from all other executions. In this dissertation, 3,000 separate injection experiments were conducted for each benchmark for each system evaluated. This gave the results a 95% confidence interval with an error margin of less than 0.75%

A drawback of this type of simulated fault injection is it can only evaluate architectural state such as the register file, instruction encoding, or program counter. The AVF of the integer register file is evaluated for each of the techniques and benchmarks to compare the reliability. Although this is not meant to encompass the reliability of the entire processor, it serves as a good estimation. The integer register file is investigated because its AVF has been shown to be more significant than other comparable structures [54, 75].

Bit flips are simulated in the register file, but this evaluation models more than just faults to the register file. A fault to the ALU unit, the result of a load, or any structure that produces data to the register can be modeled as a fault to the register after the data have been computed. While this models a larger set of faults, it does not model faults to the register bypass logic between instructions. In the fault simulation, any of the $127 \times 64 = 8128$ bits of the integer register file can be flipped in any cycle. The Intel Itanium 2 integer register file is composed of 128 64-bit registers. One register, `r0`, cannot be affected by a transient fault as it always contains the constant value zero, but the other 127 registers are vulnerable.

Simulated fault injection can be done at a variety of levels and has the same basic tradeoffs as performance modeling at various levels. Higher levels of simulation allow for longer simulations and exposure to more parts of the application, while lower levels of modeling represent the hardware more accurately but take longer to simulate. This dissertation models injections into the architecture register file, but others have simulated fault injections into a microarchitectural model [54] and an RTL model [75].

Transient faults are inherently hardware faults and thus simulation at the hardware level has greater accuracy. Simulation at this low level can model faults that occur but do not affect any persistent state. For example, a transient bit flip to a computation may be driven to the correct value before it is latched and thus will not manifest as a fault. Hardware models can also simulate faults to structures that are not visible to the architectural view, such as ALU or bypass network. However, simulation at this level is often very slow. It is only possible to simulate a small fraction of the instructions in a program. Other techniques have been proposed that do not rely on simulated fault injection, but measure all bits in a structure in a given time window and from this are able to sample the AVF of the structure [33]. These techniques also simulate at a low level and are only able to cover a small fraction of the program.

Both the fault injection and bit analysis techniques monitor how the fault propagates through the system and make conservative estimates about how the fault causes a problem. Simulations at these levels are often concerned with determining whether a fault manifests in the architectural state at all. For macro-level techniques that rely on delayed checking, errors manifesting in architectural state such as the register file are acceptable as they will be caught before they propagate to memory. Reliability evaluations via fault injection or bit analysis that solely monitor architectural propagation of a fault will not capture the detection and recovery of these software-only and hybrid techniques. This dissertation conducts a comparison of macro-reliability techniques and thus relies on fault injection at a higher level.

The reliability results presented in this dissertation are pessimistic because the simulated fault injection is done at the architectural level, the transient faults derating factors of the processor are not represented. Certain faults are not represented, such as those that would have hit the processor but would not have affected the architectural state because they were was driven to the correct value or microarchitecturally masked. Previous research has shown that various derating factors can mask up to 85% of transient faults in the pipeline of the processor using an RTL mode [75]. The abstraction of derating factors is conducted in order to focus on the architectural factor of protection and to remove the process and environmental factors of the overall reliability, but it is an important consideration when comparing the reliability of the presented techniques.

Chapter 6

Evaluation of Software and Hybrid Protection

This chapter provides a quantitative evaluation of the techniques presented thus far. The SWIFT, SWIFTR, CRAFT, and CRAFTR techniques are evaluated in terms of performance in Section 6.1 and reliability in Section 6.2. These two sections compare only the versions of these techniques without any precise configurations. Although a major advantage of these methods is the ability to be easily configured to tradeoff performance and reliability, that analysis is delayed until Chapter 8. Section 6.3 relates these evaluations to previously proposed techniques.

6.1 Performance

The section compares the performance of the various reliability techniques. Performance in these graphs is given relative to the unprotected applications. In these performance graphs, the y-axis is the execution time of the given technique normalized to the unprotected version. Each bar on the x-axis represents a different application that is evaluated. The last bar of the performance graphs represents the Geometric Mean of all of the applications. Because performances and MWTF are normalized to the unprotected application, this dis-

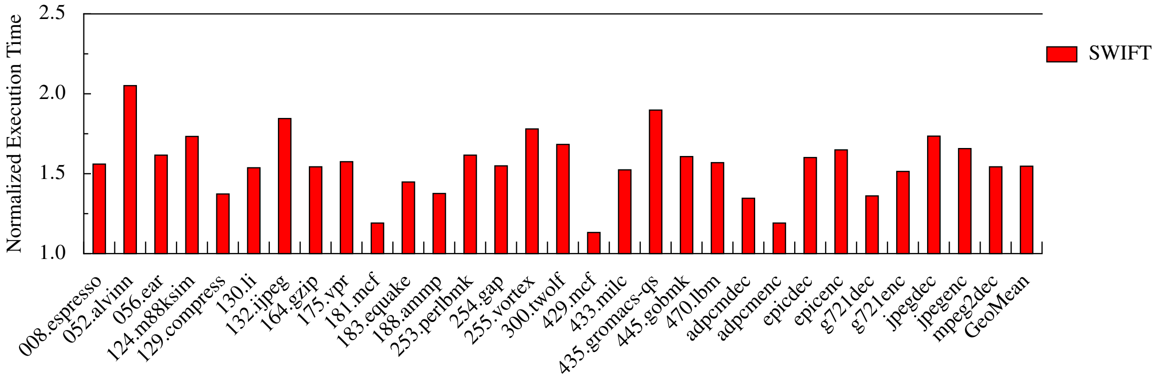

Figure 6.1: Performance of SWIFT Normalized to the Unprotected Versions

sertation uses the Geometric Mean as it gives equal weight to each benchmark although their runtimes vary.

The results in Figure 6.1 show the normalized execution times for the software-only detection technique, SWIFT. The Geometric Mean illustrates that on average, the reliability cost in terms of performance is about 50% (specifically, a 1.54x increase in execution time).

If the unprotected program were run twice, the normalized execution time would be exactly 2.00. Additional code would need to be executed to compare and validate the two program outputs, thus the overall degradation would be much greater than 2.00. The 1.54x normalized execution time of SWIFT indicate that it is exploiting the unused processor resources present during the execution of the baseline program. The benchmark `429.mcf` shows only a 1.13x increase in normalized execution time. This benchmark is known to be mostly limited by memory accesses. The duplicated computations and the verification code of SWIFT are able to be executed off of the critical path.

Benchmarks like `jpegdec`, `jpegenc`, `052.alvinn`, and `435.gromacs` are more computationally intensive and thus have less spare resources. When instructions for reliability are added, performance is reduced. However, notice that only one benchmark requires more than a 2.0x normalized execution time (which is `052.alvinn`, which has the worst normalized execution time of 2.05x) when adding a full copy of the computation and verification code. This means that two versions of the application plus the verification

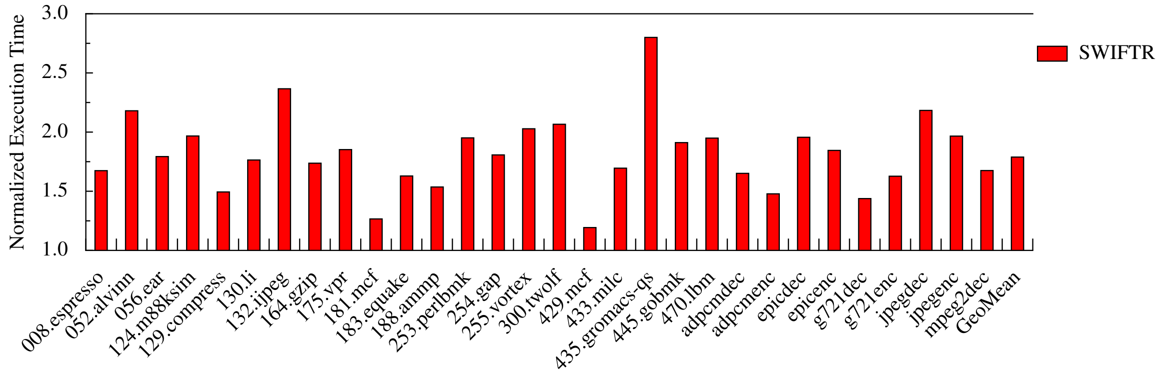

Figure 6.2: Performance of SWIFTR Normalized to the Unprotected Versions

instruction are able to be executed in the spare resources of the processor.

The results in Figure 6.2 show the normalized execution times for the software-only recovery technique, SWIFTR. The Geometric Mean illustrates that on average, the reliability cost in terms of performance is 78%, or a 1.78x increase in execution time. In SWIFT, which has two copies of the computation, the normalized execution time is less than 2.0x and in SWIFTR, which has three copies, the normalized execution time is less than 3.0x. For all but five benchmarks, the normalized execution time of SWIFTR is still less than 2.0x. Again, like SWIFT, SWIFTR is able to use the unused resources to do the redundant computation in parallel. The extra copies of the code are independent, which is necessary for reliability. That also means they can be scheduled at the same time as the original copies. Similar to SWIFT, the worse performing benchmarks are benchmarks like `jpegdec`, `jpegenc`, `052.alvinn`, and `435.gromacs` which are computationally intensive and have less spare resources while those that incur a smaller performance penalty are benchmarks like `429.mcf`, which are limited by memory accesses.

Figure 6.3 shows the SWIFT and SWIFTR performance on the same graph to illustrate the transition from detection to recovery. The unprotected application is abbreviated NOFT (no added fault tolerance). For `435.gromacs`, the transition from unprotected to detection-only incurs about the same cost as from detection-only to recovery. SWIFT normalized execution time is 1.89x and SWIFTR is 2.80x. Although still less than 2.0x and

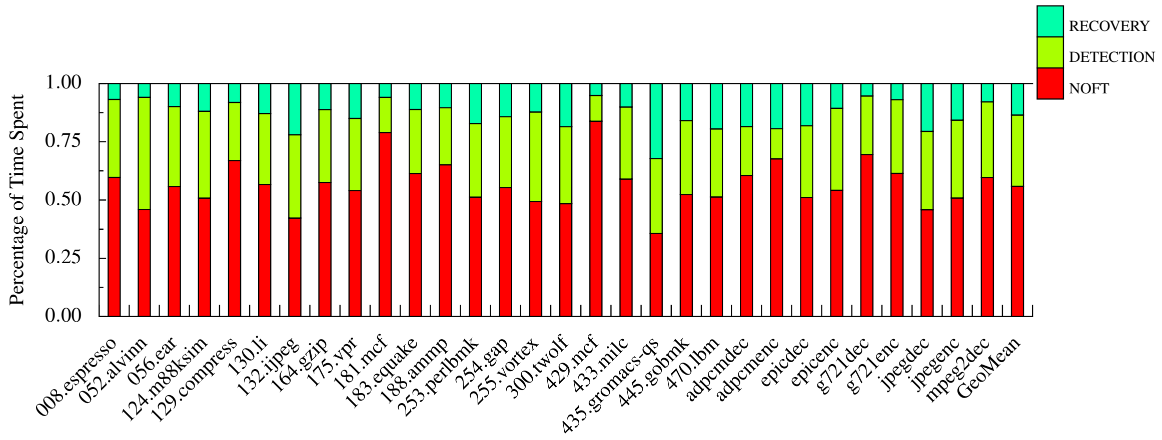

Figure 6.3: Performance of Software-Only Techniques Broken Down by Percent of Execution Time

3.0x, respectively, `435.gromacs` in unable to hide the cost of the extra reliability instructions. For most benchmarks, the incremental cost of detection is more than the incremental cost of recovery. The Geometric Mean for detection is 1.54x but the Geometric Mean for recovery is 1.78x, which is only 0.24x more than detection. For certain benchmarks like `g721dec`, `g721dec`, `mpeg2dec`, `008.espresso`, and `429.mcf`, the cost of reliability is less than 10% of the execution time of the application. In these cases, once the user has decided to add reliability, the cost to add recovery is so minor that adding recovery is always beneficial. In these cases, protection with recovery is an efficient choice.

The incremental cost of reliability is less than the incremental cost of detection because the detection instructions are not always using idle resources. The detection code is executed on the critical path leading to increases in execution time. When the recovery code is executed, it is independent of both other versions (the original and the first redundant). The third version is able to be more efficiently scheduled using both the leftover idle resources from the original version and the spare resources from the execution time increase due to the second version.

Figure 6.4 shows the performance evaluation for the hybrid detection-only and recovery techniques. Figure 6.5 shows a breakdown of the performance costs by the level of reliability techniques. Relative to each other, the hybrid techniques show patterns that are

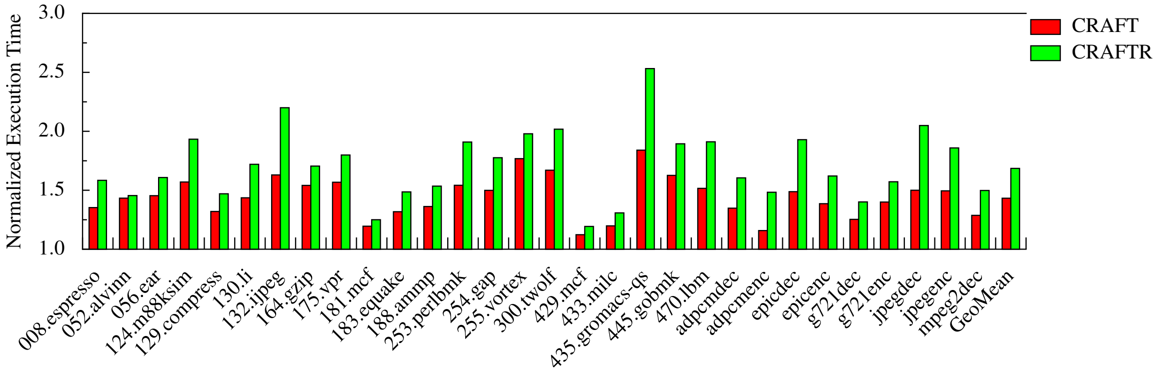

Figure 6.4: Performance of Hybrid Techniques Normalized to the Unprotected Versions

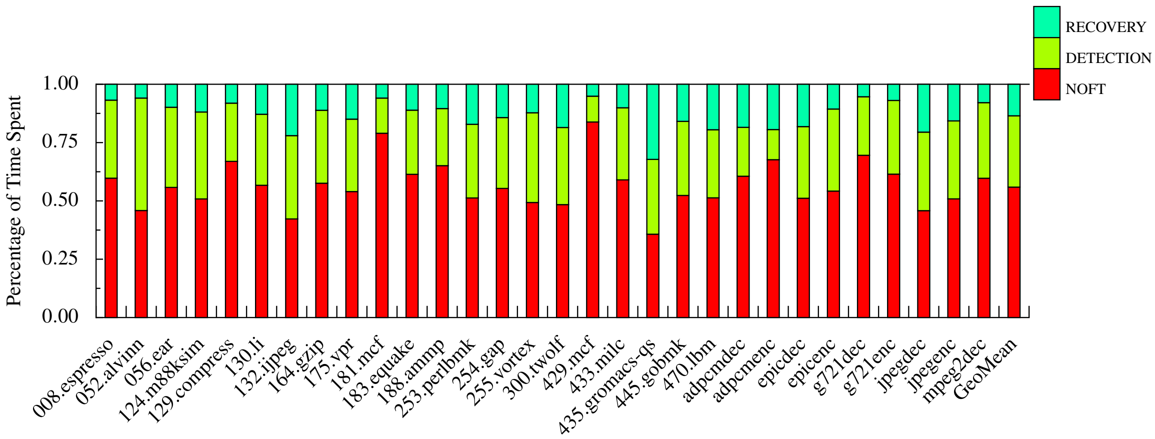

Figure 6.5: Performance of Hybrid Techniques Broken Down by Percent of Execution Time

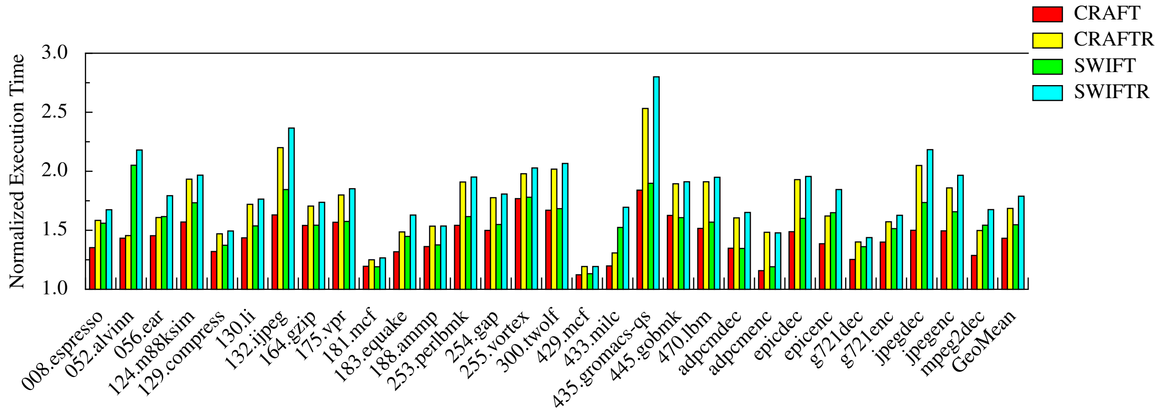

Figure 6.6: Performance of Software-Only and Hybrid Techniques Normalized to the Unprotected Versions

very similar to the software-only techniques. This is not surprising as the hybrid changes, while important for reliability and improving performance, were minimal.

Figure 6.6 depicts the normalized performance for all four techniques. The performance of the hybrid techniques is better than the corresponding software-only techniques in all cases, as they remove the checking instruction by offloading the validation to hardware. This removal of validation instruction creates more flexibility. In the software-only techniques, the load and store instructions are synchronizing points, so both versions of the computation had to be completed before one version could execute. In the hybrid implementation, the original and reliable versions are independent even through loads and stores because the hardware implements the comparisons. This results in a performance improvement compared to the software-only techniques.

6.2 Reliability

This section details the reliability evaluation of the software-only and hybrid techniques. As described in Chapter 5, there are two components to reliability evaluation. Section 6.2.1 evaluates the techniques in terms of the AVF and Section 6.2.2 evaluates them using MWTF.

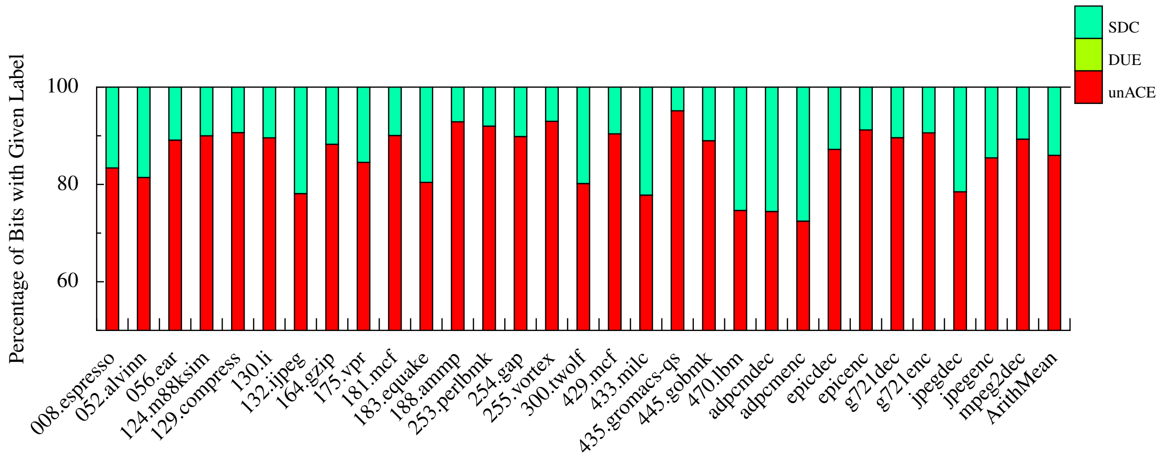

Figure 6.7: Reliability with No Fault Protection

6.2.1 Architectural Vulnerability Factor

As mentioned in the previous chapter, this dissertation evaluates the AVF via fault injection into the register file. A simulated fault toggles one bit in a register uniformly distributed over the time of the application, and the application run to completion. The output of the execution is compared to the baseline versions and the bit is determined to be unACE, DUE, or SDC based upon the result.

First, the reliability of the unprotected applications is shown in Figure 6.7. On average, 85.97% of the faults injected do not cause any noticeable errors and 14.03% of the injections result in incorrect output. As there is no detection in the baseline application, 0.00% of the faults result in DUE.

There is variety in the native application reliability. `470.lbm`, `adpcmdec`, and `adpcmenc` have the lowest native reliability at 74.46%, 74.44%, 72.46%, respectively. `435.gromacs` has the highest at 95.14%. This variation is important when deciding what to protect and will be illustrated in detail in Chapters 7 and 8. For now, this will be used as a baseline when considering how the reliability technique benefits the application.

Figure 6.8 shows the reliability of the SWIFT software-only technique contrasted with the unprotected application. The AVF_{SDC} is greatly reduced, as the redundant computation ensures that a fault will not affect both values and the checking is done before the values

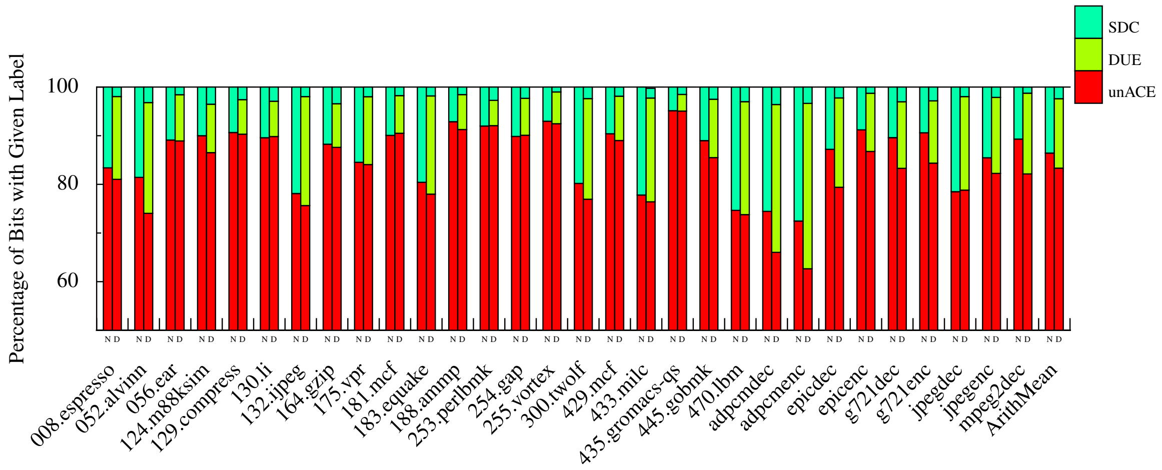

Figure 6.8: Reliability of SWIFT Compared to the Unprotected Versions

propagate and harm the system.

Figure 6.8 also shows a significant amount of AVF$_{DUE}$. This is the portion of time that the fault is detected and execution is halted. When comparing NOFT to SWIFT, a large portion of the SDC bits are converted into DUE bits. This is expected because the faults that would have caused application failure are detected by the redundant versions and the application is halted.

Figure 6.8 also illustrates that some of the unACE bits of the NOFT techniques are converted into DUE bits. This means that some of the faults that were injected would not have caused any visible errors, but were detected by the system and the application was halted. This could happen if, for instance, a faulty value was about to be written to memory but would never be read from memory. The erroneous value would not have caused incorrect execution, but the detection system would halt the program. In this implementation, memory is considered outside the SoR thus all values written to it are verified. Knowing whether the faulty value would cause a problem is impossible and the system is conservative in this situation and detects the faults. As reviewed in Chapter 3, detection is delayed until memory instructions; values that are wrong (in the register file) but do not affect the output do not halt the program.

Although possible to create a software-only system that only checks the program at I/O memory accesses, this system would duplicate memory. The duplication would be problematic because memory accesses are often the limiting factor in execution. The system would also remove checks at load and store operations that do not access I/O. The lack of validation is a problem because if a store that should write to non-I/O memory has an invalid address, incorrect data will be written to the wrong location. The store instructions may corrupt the other version's data or may write to memory I/O.

The Arithmetic Mean of the reliability of the benchmarks shows a reduction in AVF$_{SDC}$ from 13.56% to 2.39%. However, SWIFT does not reduce the percent of SDC bits to zero. As mentioned in Section 3.4, there are certain windows of vulnerability that leave the application unprotected, such as between validation and use of the data in loads, stores, and control flow instructions. A nonzero AVF$_{SDC}$ is also due to certain implementation decisions. In order to maintain the calling convention, argument passing is not duplicated. Before entering a function, argument registers are checked against their duplicate counterparts to ensure their values match. Then, after entering the function, the parameters are copied into duplicate registers for use by the subsequent redundant code. A transient fault in one of the parameters to the function after the check but before the copy will lead to an undetected fault. Another source of SDC bits arises from exceptions that terminate the program. For example, if a fault occurs in data that feed the denominator of a divide instruction, that fault may create a divide-by-zero exception and terminate the program. The techniques do not validate the source operands of divide instructions, only data being written to memory or affecting control flow.

Figure 6.9 shows the reliability in terms of AVF for the software-only recovery technique, SWIFTR. Because there are three versions executing simultaneously, when an error is detected, it is able to recover and continue and as such, the AVF$_{DUE}$ is reduced to 0.00%. Comparing to the unprotected version, the SWIFT version has a smaller percentage of bits labeled unACE due to eager validation. The SWIFTR version has a much larger percentage of bits labeled unACE, and this is exactly the same percentage of SDC bits that is removed.

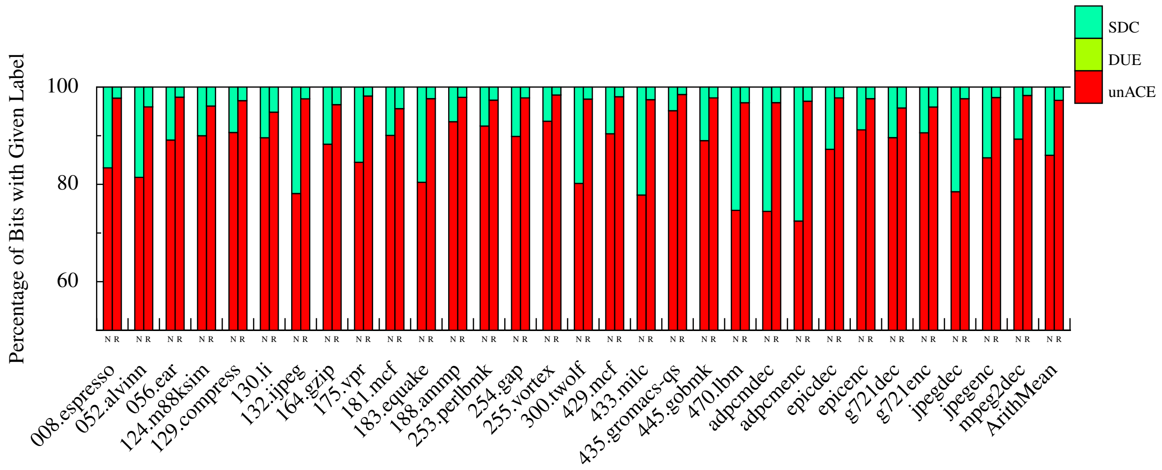

Figure 6.9: Reliability of SWIFTR Compared to the Unprotected Versions

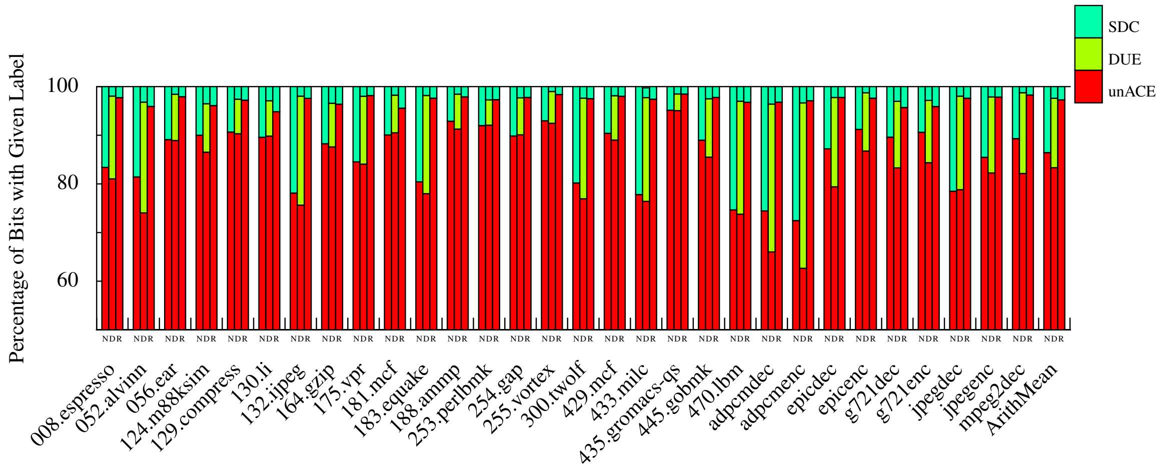

Figure 6.10: Reliability of SWIFT and SWIFTR Compared to the Unprotected Versions

Again, there are still some windows of vulnerability that keep the techniques from having a 0.00% AVF$_{SDC}$.

Figure 6.10 illustrates the reliability in terms of AVF for both the software-only techniques. The recovery technique increases the percentage of unACE bits the most, while the percentage of SDC bit is reduced the most by the detection-only technique. This is due to the fact that the recovery version has more instructions to execute and the corresponding time of the windows of vulnerability is larger. The recovery version also has a greater number of windows of vulnerability. Although only a small amount, this is noticeable by the small amount of SDC increase in some application like `130.li`. However, when only

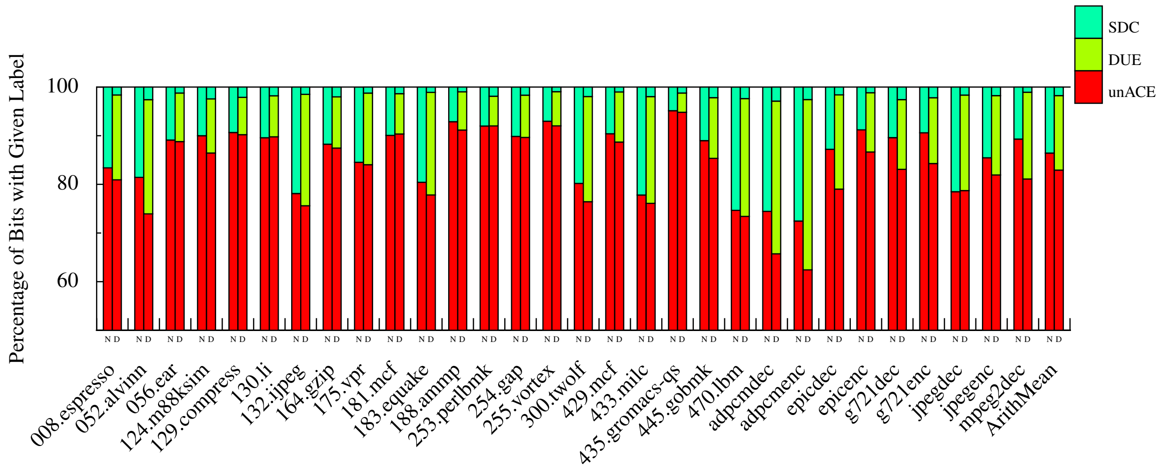

Figure 6.11: Reliability of CRAFT Compared to the Unprotected Versions

considering the percent of faults that still allow the program to execute, SWIFTR is more reliable, but costs more in terms of execution time. The tradeoff between performance and reliability will be analyzed when MWTF is evaluated in Section 6.2.2.

Figure 6.11 shows the AVF of the hybrid detection-only CRAFT technique compared to the native application. Compared to the baseline application, the CRAFT technique has the same trend as the SWIFT technique. The AVF_{SDC} percentage is reduced and replaced with detection. Also, the percentage of unACE bits is noticeably reduced for the same reasons that the software-only technique has a smaller percentage of unACE bits.

Although there is no window of vulnerability around memory instruction in the CRAFT technique, there are still windows of vulnerability around control flow and calling convention instructions. Also, similar to the software-only technique, signals that terminate the application like divide-by-zero can cause SDC. However, there is another source of SDC unique to CRAFT that arises from exceptions that terminate the program – segmentation faults. Recall from Section 4.3 that the code which uses the LVQ does not check the address before issuing the first load instruction, leaving validation to the hardware structure. Since the first load is sent to memory directly, while the second is serviced from the LVQ, a fault in the address of the first load may cause a segmentation fault. This increases the SDC of techniques which incorporate the LVQ.

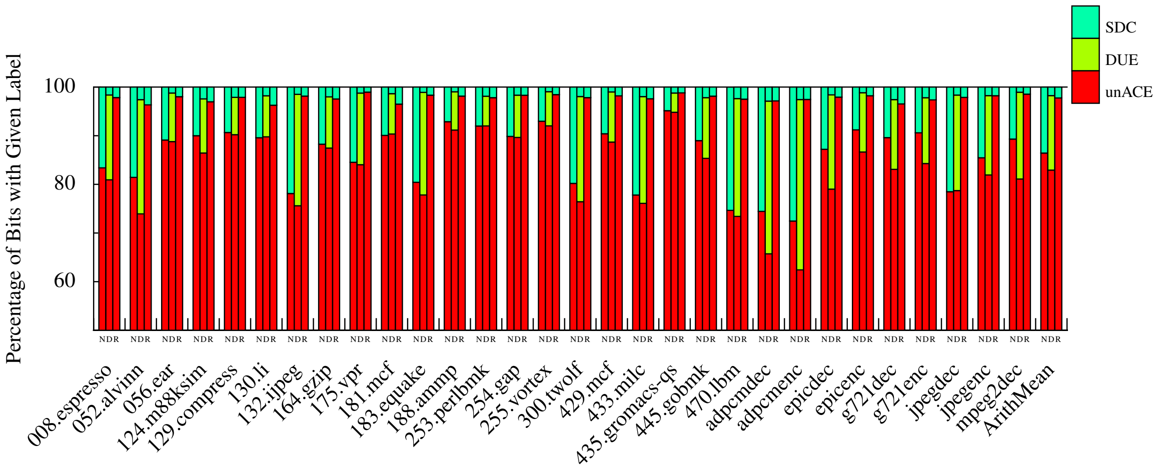

Figure 6.12: Reliability of CRAFT and CRAFTR Compared to the Unprotected Versions

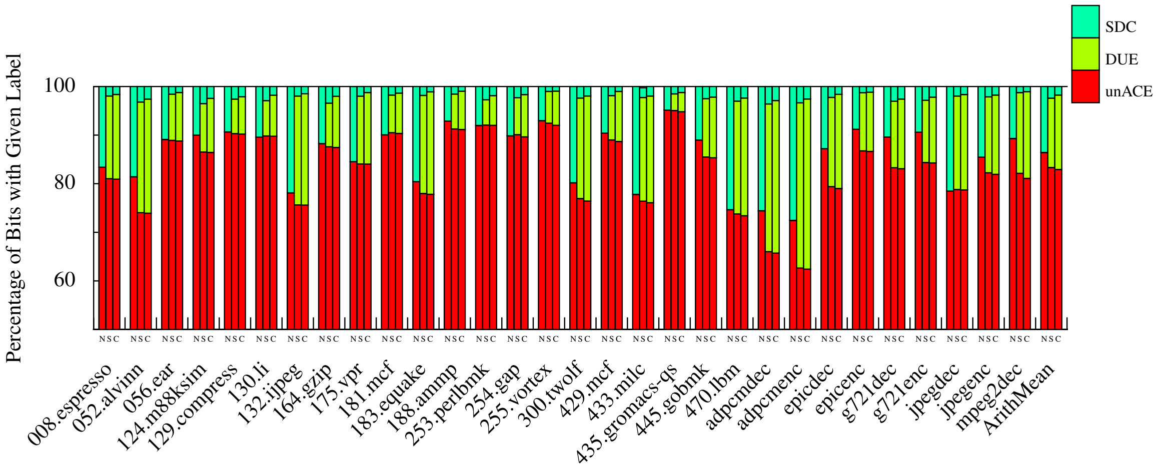

Figure 6.13: Reliability of the Detection-Only Techniques Compared to the Unprotected Versions

The results of the CRAFT and CRAFTR techniques follow the same trends as the SWIFT and SWIFTR techniques. Figure 6.12 shows the AVF of the recovery version of the hybrid technique, CRAFTR, in addition to the detection-only technique and unprotected application. Similarly to SWIFTR, CRAFTR has replaced the DUE bits with unACE bits. It is able to recover in the situations when CRAFT aborts the program and signals fault detection. Again, the recovery technique has the highest percentage of unACE bits, but the detection technique has the lowest AVF$_{SDC}$.

Figures 6.13 and 6.14 show a comparison of the four techniques analyzed thus far in

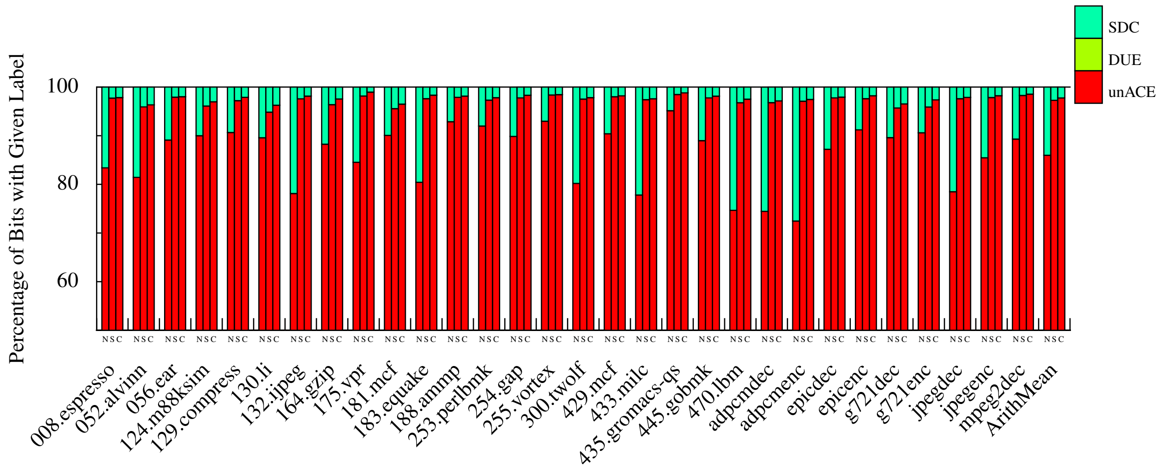

Figure 6.14: Reliability of the Recovery Techniques Compared to the Unprotected Versions

terms of AVF. These graphs illustrate the increase in reliability moving from the software-only to the hybrid technique. The hybrid techniques offer benefits in terms of both reliability and performance. Given that these are usually the two main tradeoffs, these advantages make hybrid techniques preferable. However, they do have a minimal hardware cost that may limit the applicability of the hybrid techniques. The software-only techniques, although less reliable with worse performance than the hybrid techniques, are immediately applicable.

6.2.2 Mean Work To Failure

This section describes the reliability of the techniques from the perspective of another metric. By combining the performance measurements from Section 6.1 and the AVF measurements from Section 6.2.1, it is possible to compute the MWTF. Recall from Section 5.2.2 that MWTF combines both the increased reliability in terms of reduction in failure due to soft errors and the increase in runtime which increases the exposure to soft errors. AVF is often reported as a measurement of reliability, but MWTF incorporates both the AVF and performance to compute of the efficiency of a reliability technique. MWTF also creates a level playing field for comparing multiple techniques that alter both execution time and

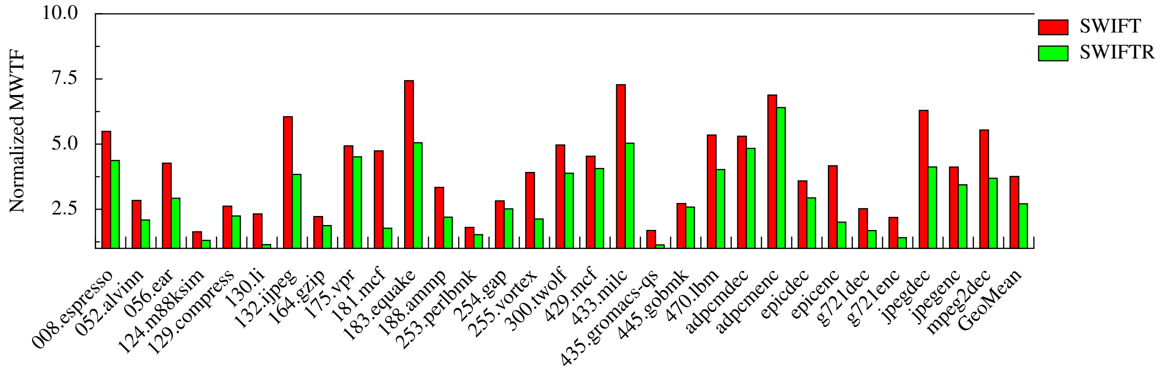

Figure 6.15: Normalized MWTF of SWIFT and SWIFTR where Failure is SDC

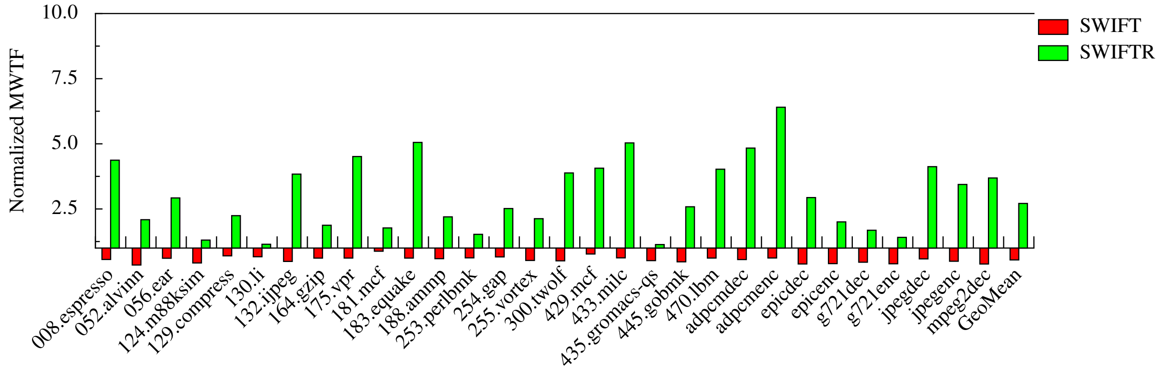

Figure 6.16: Normalized MWTF of SWIFT and SWIFTR where Failure is SDC and DUE

reliability.

Figures 6.15 and 6.16 show the nMWTF for both software-only techniques. The MWTF in both figures is normalized to the unprotected baseline. Notice that the x-axis is positioned at a nMWTF of 1.0 to clearly show how the reliability techniques compare to the unprotected application. Also notice that between Figures 6.15 and 6.16, the nMWTF of the SWIFTR technique does not change, but the nMWTF of SWIFT changes dramatically.

Figure 6.15 shows the nMWTF where failure is defined as any time the application executes and produces incorrect output (in Figure 6.15, failure includes only SDC). In this case, the SWIFT and SWIFTR nMWTF for each benchmark is greater than 1.0. Some benchmarks like `124.m888ksim` and `435.gromacs` are very similar to the unprotected application with an nMWTF of 1.63x and 1.65x, respectively, for the detection technique and 1.30x and 1.14x, respectively, for the recovery technique. The benefit due to the increase in reliability for these benchmarks is about the same as the cost due to the increased execution time. Considering this definition of failure, the nMWTF of SWIFTR is less than that of SWIFT. This is not surprising in that the performance cost of SWIFT is less than that of SWIFTR, and the SWIFT technique reduces the AVF_{SDC} by at least the same amount as (and often more than) the SWIFTR technique. SWIFT does a better job than SWIFTR with this definition of failure on both metrics that comprise nMWTF.

Figure 6.16 shows the nMWTF where failure is defined as any time the application does not run to completion with the correct output. In Figure 6.16, failure includes SDC and DUE. In this case, the SWIFT nMWTF is always less than 1.0, thus when comparing both performance and reliability, this reliability technique is not worth implementing. Recall that SWIFT's increase in the percentage of DUE bits reduces the percentage of both SDC and unACE bits compared to the original application. This reduction in percentage of unACE bits, combined with the increase in the execution time of SWIFT compared to the unprotected application, indicates that SWIFT does worse than the unprotected code on both metrics that comprise nMWTF using this definition of failure.

In both Figure 6.15 and Figure 6.16 the nMWTF of the SWIFTR code remains the same because SWIFTR has no program halts from detection and therefore no DUE. For both definitions of failure, and therefor in both figures, the nMWTF of SWIFTR is greater than 1.0. For some benchmarks like `adpcmenc`, the nMWTF is as high as 6.40x and for others, like `435.gromacs`, is only 1.13x. For benchmarks with a low nMWTF, the unprotected application is naturally resilient and the addition of the SWIFTR (or SWIFT) transformation does not provide much added reliability. For `435.gromacs`, the percentage of unACE bits for the unprotected application is 95.14%. The SWIFTR transformation does not have much room to improve the reliability. SWIFTR is able to bring the unACE percentage to 98.46% but at a cost of 180% increase in execution time (normalized runtime of 2.80x). This is in contrast to a benchmark like `adpcmenc` which has an nMWTF of 6.40x. The unprotected application has 73.46% of bits labeled as unACE and that is

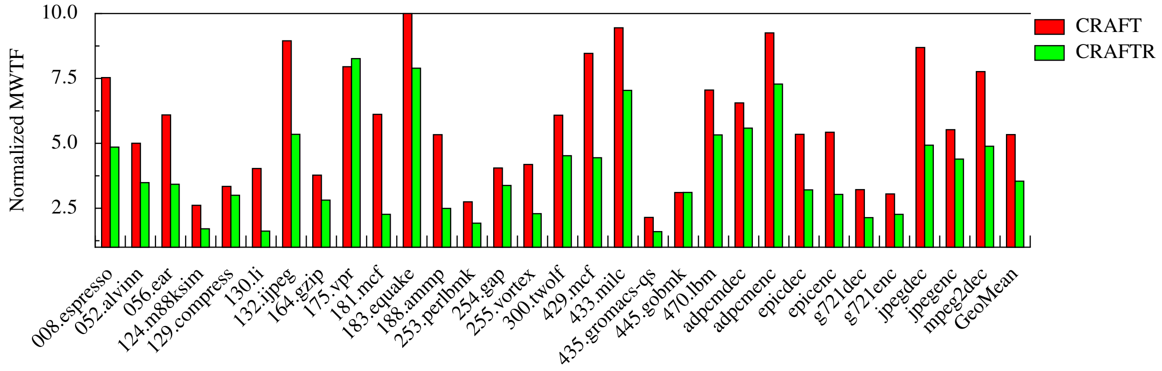

Figure 6.17: Normalized MWTF of CRAFT and CRAFTR where Failure is SDC

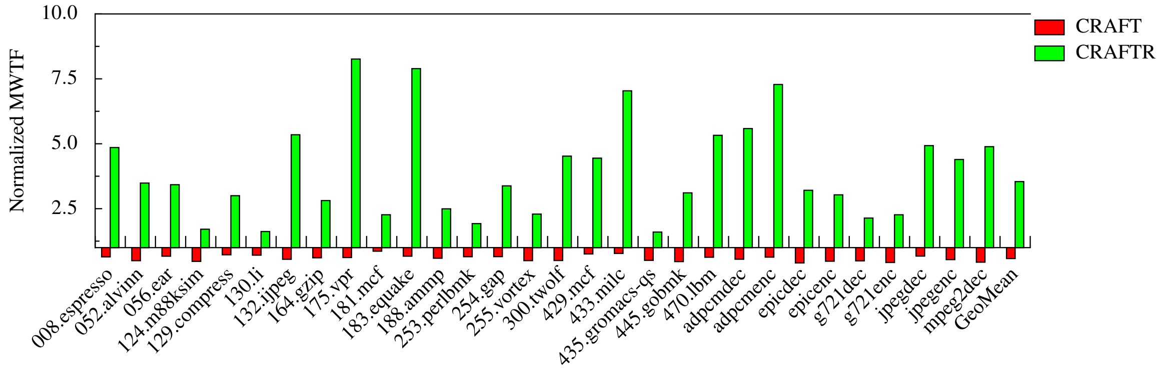

Figure 6.18: Normalized MWTF of CRAFT and CRAFTR where Failure is SDC and DUE

increased to 97.09% with the SWIFTR transformation. This is done at a cost of 47% increase in execution time (normalized runtime of 1.47x). The large increase in the percentage of unACE bits and small increase in normalized execution time results in a large nMWTF. The SWIFTR transformation provides less overall reliability for `adpcmenc` than for `435.gromacs` (97.09% compared to 98.46%), but provides a large difference compared to the unprotected versions for `adpcmenc` and does so with less performance cost than `435.gromacs` (1.47x compared to 2.80x).

Figures 6.17 and 6.18 show the nMWTF for the CRAFT and CRAFTR techniques. The nMWTF for these techniques follows the same trend as for the software-only techniques. Depending on the definition of failure and whether it includes program halts due to detection, the CRAFT or the CRAFTR technique produces a larger nMWTF. Like the software-only techniques, in which detection is considered part of the definition of failure,

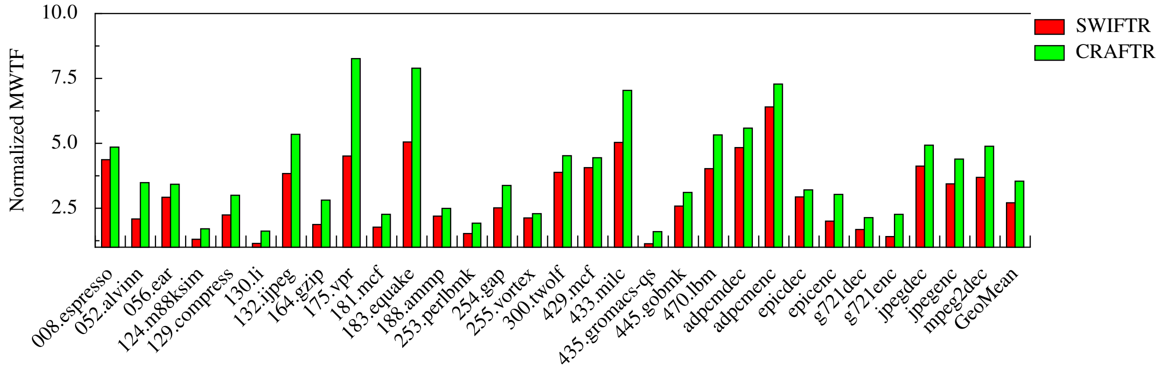

Figure 6.19: Normalized MWTF of SWIFTR and CRAFTR where Failure is SDC

no applications with the CRAFT detection-only technique have an nMWTF greater than 1.0. Despite the reduced failure rate and improved performance of CRAFT over SWIFT, for this definition of failure the reliability increase of the detection-only hybrid technique is not worth the performance cost.

The hybrid nMWTF and the software nMWTF follow the same patters but differ in magnitude. Recall that the hybrid techniques reduce the windows of vulnerability around store and load instructions, thereby increasing reliability. The hybrid techniques also allow for greater scheduling flexibility because the code contains fewer synchronizing points. This allows for improved performance relative to the software-only techniques. The hybrid techniques improve both the performance and reliability components of nMWTF and thus realize a greater nMWTF than the software-only techniques.

For the recovery techniques, Figure 6.19 shows that the average nMWTF for the software-only technique is 2.71x and the average nMWTF for the hybrid technique is 2.54x. This is expected as the hybrid technique reduces the performance cost and increases reliability. As can be seen in Figure 6.19, each application evaluated has a larger nMWTF for the hybrid technique than for the software-only technique. These results reinforce the conclusion that if the hardware for the hybrid technique is available, it is the most beneficial technique to use.

6.3 Comparison to Hardware-Only Techniques

Hardware-only redundancy techniques, such as lockstepping and RMT, typically have a smaller performance penalty than software-only approaches. The RMT hardware technique has been shown to suffer a 32% slowdown when compared to a non-redundant single-threaded core and a 40% slowdown when compared to a non-redundant multi-threaded core [32, 50]. These numbers correspond to normalized execution times of 1.47x and 1.67x, respectively. Lockstepping techniques generally have worse performance than RMT and a larger hardware cost, because techniques that execute in lock step have to validate after each instruction rather than validating only when data can potentially corrupt memory as in RMT (and hybrid) techniques. Although the techniques in this dissertation were evaluated on an Intel Itanium 2 processor and the RMT implementation on a Compaq EV8 processor, a comparison of the results is still meaningful and instructive. Normalized execution times for CRAFT are similar to those for RMT. The hybrid techniques do benefit from the large number of architectural registers in the Intel Itanium 2 architecture. The duplication of code in the software level puts additional pressure on the architectural register file. Hardware techniques like RMT are implemented at the microarchitectural level and have access to the physical registers, which are often more abundant than architectural registers.

Chapter 7

Software Modulated Fault Tolerance

The software-only and hybrid techniques presented in this dissertation thus far increase the application reliability without adding excessive execution time overhead. The software-only and hybrid techniques are also advantageous because unlike other macro-redundancy techniques, they are easily amenable to fine-grain tradeoffs between performance and reliability. The reliability benefit of these techniques can be easily configured to match the requirements of the system. Native reliability of an application is one of the three characteristics used when determining the most efficient configuration of reliability to use, and is explained in Section 7.1. To find the optimal reliability configuration, the modulated protection technique also monitors two other characteristics: changes to both the performance and reliability when redundancy is added to an application. Sections 7.2 and 7.3, respectively, describe differences in how the applications respond in term of performance cost and reliability benefit when redundancy is added. Section 7.4 explicates the process of analyzing the performance and reliability responses along with the native reliability of the application to find an optimal reliability configuration. Finally, Section 7.5 reviews existing research on partial and configurable transient fault protection.

The configurable fault tolerance technique proposed in this dissertation is Software Modulated Fault Tolerance (SMFT). The reliability configurations and tradeoff are dictated by software decisions. During the code transformation process from an unprotected to protected application, the compiler system decides how to modulate the fault protection of each region. SMFT is easily applied to the SWIFT, SWIFTR, CRAFT, and CRAFTR techniques because of the redundancy in software.

7.1 Variations in Failures for Unprotected Code

This section details the inherent variations in reliability of unprotected applications. Figure 6.7 shows the variations in application reliability for the range of benchmarks and illustrates that the natural percentage of unACE bits in the application ranges from 72.46% for `adpcmenc` to 95.14% for `435.gromacs`.

These variations form one of the key characteristics that are exploited when protecting an application with configurable fault tolerance: certain applications are naturally more reliable than others and may require less protection. For a given reliability specification, some applications may be already within the reliability threshold, and thus it would be unnecessary to apply the techniques to those applications. The extra room in the budget can be spent on more unreliable applications or discarded for better performance.

Just as individual applications have a wide variety of natural fault resilience within an application suite, individual regions within an application have a wide range of reliability. Figures 7.1, 7.2, 7.3, and 7.4 show a breakdown of the region reliability for the application `jpegdec`, `300.twolf`, `433.milc`, and `445.gobmk`, respectively. Each figure shows the functions that comprise 95% of the application runtime and the corresponding percentage of unACE bits. The percentage of unACE bits for the overall application is also shown in each figure. These figures illustrate that the application's reliability varies across the different regions of the program. While `jpegdec` may have an average percentage of unACE bits of 78.49%, significant regions within the benchmark range from 94.97% to 56.77%.

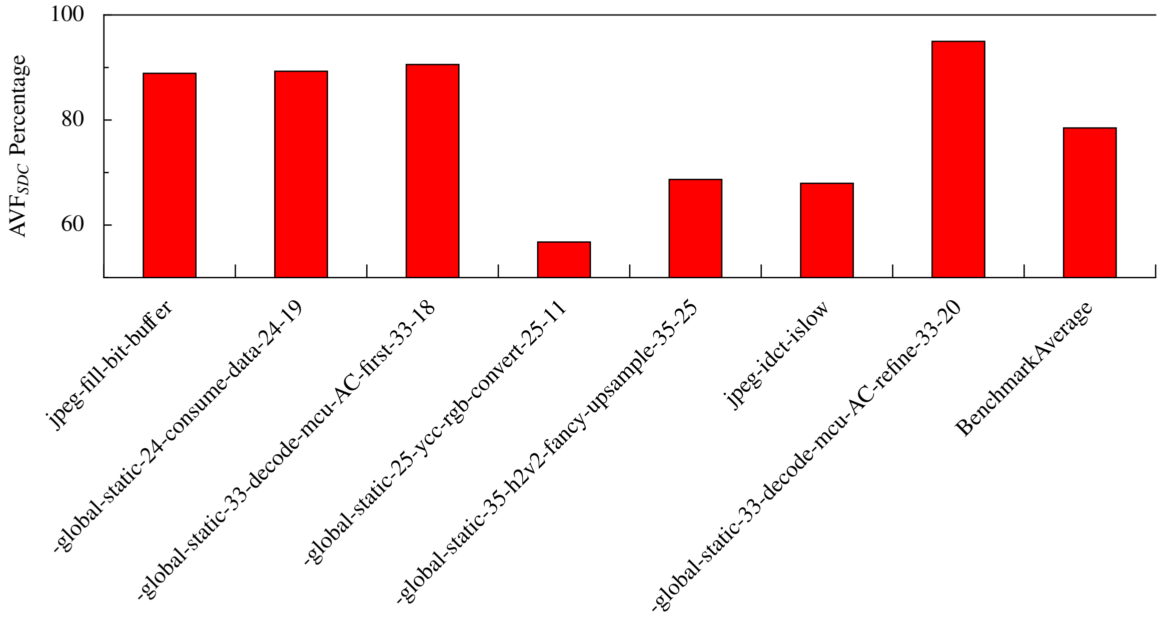

Figure 7.1: Variations in Natural Reliability within the `jpegdec` Benchmark

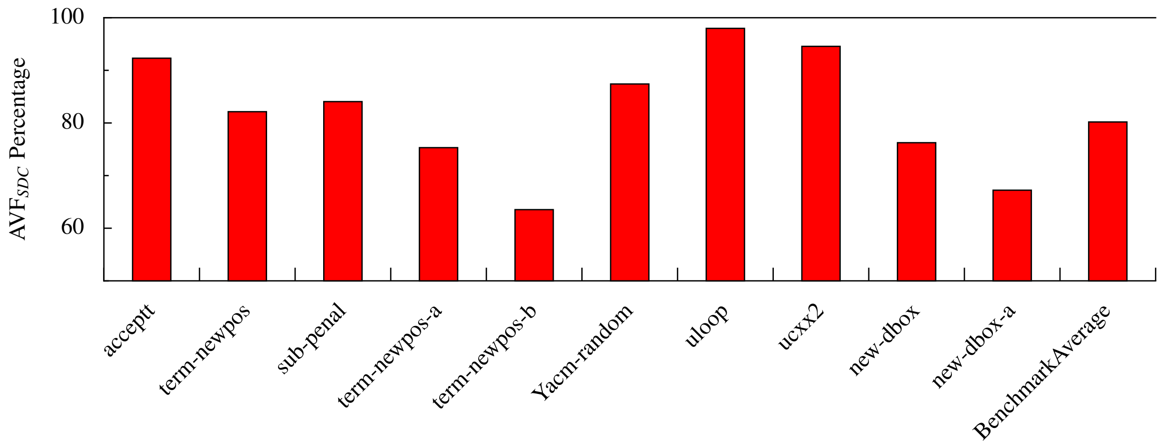

Figure 7.2: Variations in Natural Reliability within the `300.twolf` Benchmark

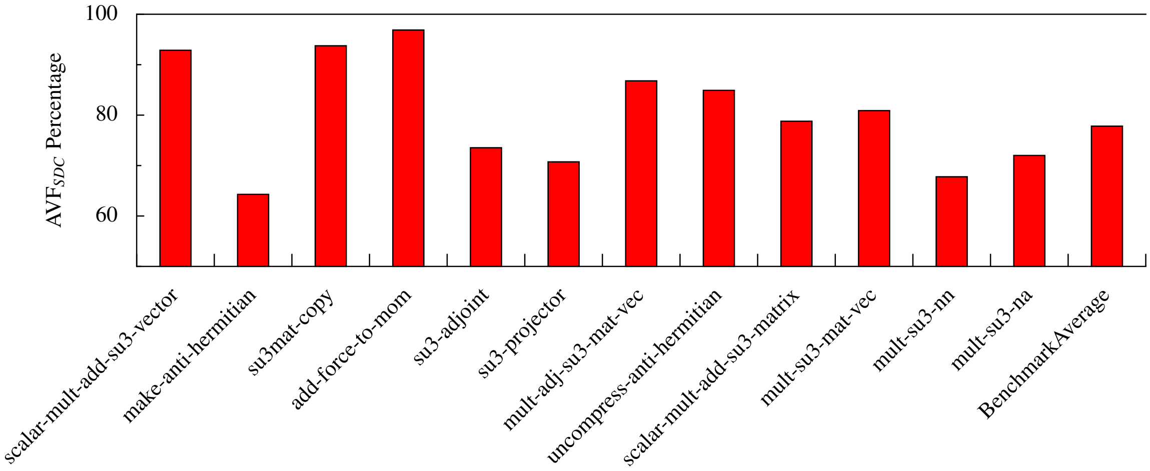

Figure 7.3: Variations in Natural Reliability within the `433.milc` Benchmark

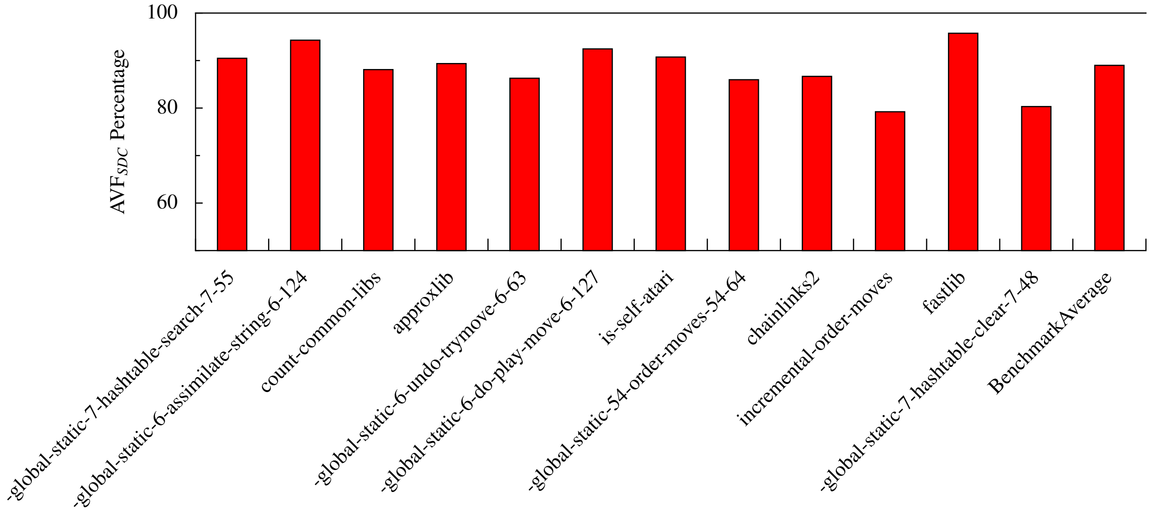

Figure 7.4: Variations in Natural Reliability within the `445.gobmk` Benchmark

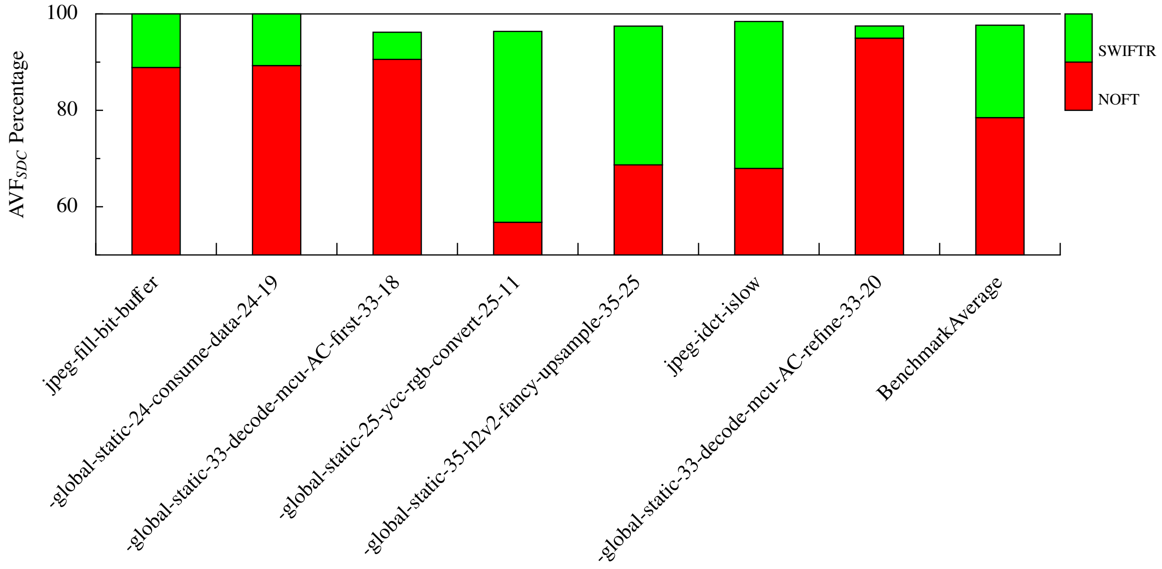

Figure 7.5: Variations in Reliability Responses within the `jpegdec` Benchmark

Variations at the region or application level exist for a variety of reasons. The inherent nature of the data access patterns can create higher or lower reliability. For example, data that live in the register file for a long time are more susceptible. Other researchers have noticed similar patters in other hardware structures and used this information to do intelligent flushing and increase reliability [76]. As mentioned in Chapter 6, bit masking [33] and programming factors such as Y-Branches [74] can also have significant effects on hiding faults.

7.2 Variations in Reliability Responses

As there is a wide variety in natural reliability across and within applications, there is a similar range in reliability after redundancy techniques are added. The reliability after the technique has been applied compared to the unprotected application is referred to as the *reliability response*. Figures 6.9 and 6.12 from the previous chapter illustrate the range in reliability responses after the software-only and hybrid techniques are applied. A variety in magnitude of reliability responses exists for all techniques including both software and

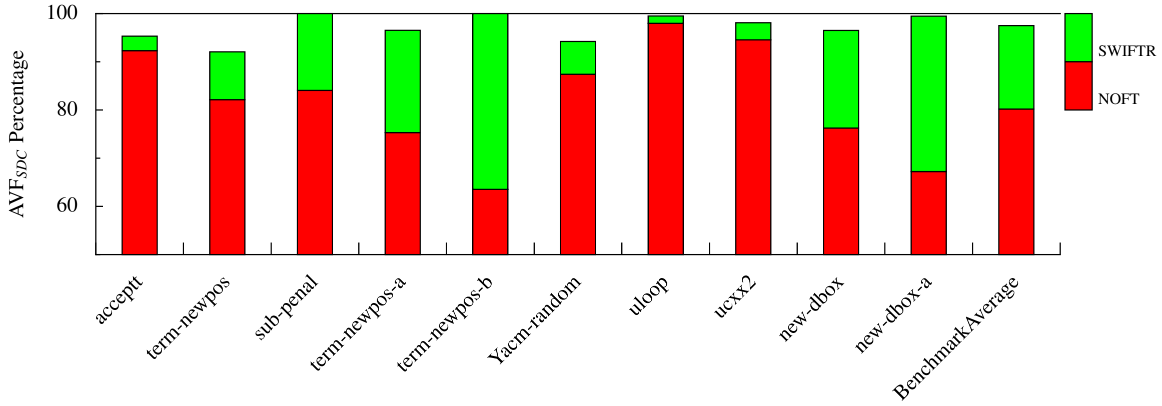

Figure 7.6: Variations in Reliability Responses within the `300.twolf` Benchmark

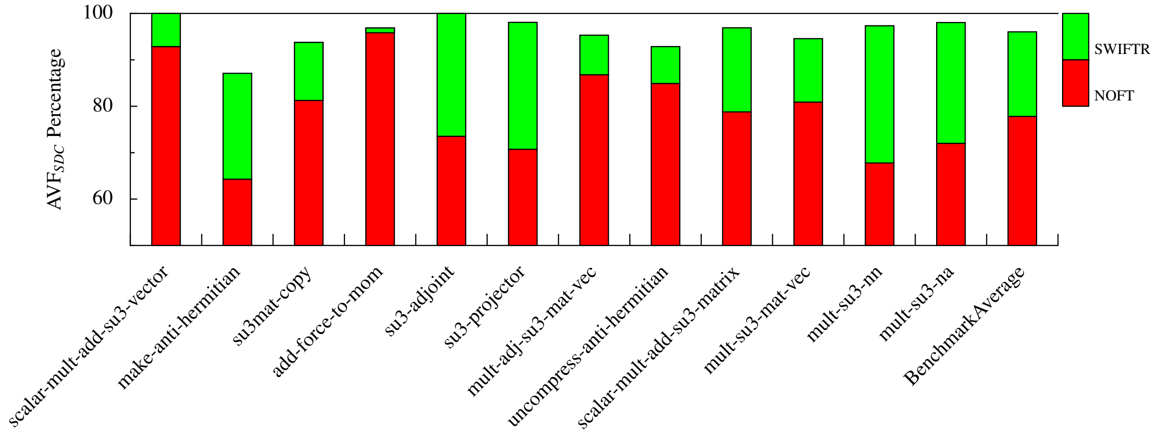

Figure 7.7: Variations in Reliability Responses within the `433.milc` Benchmark

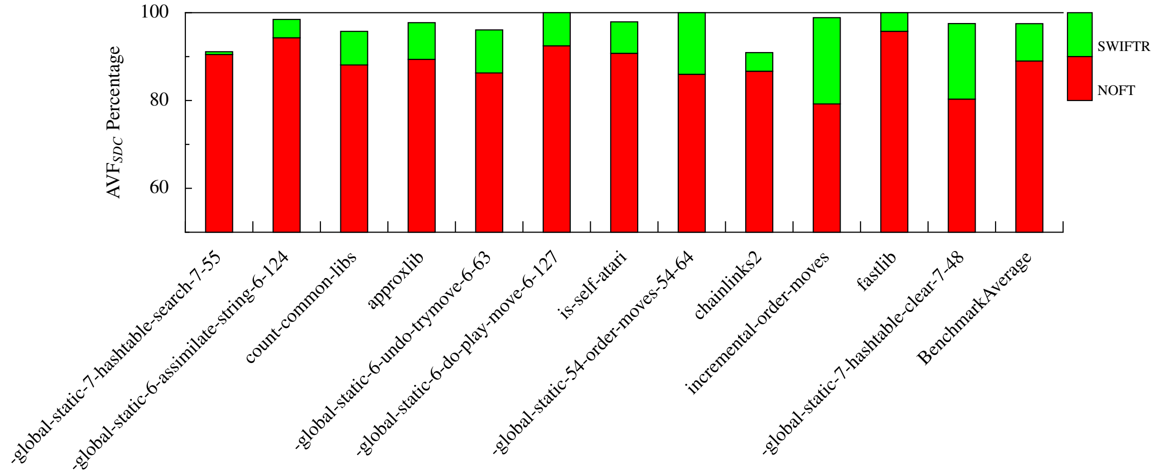

Figure 7.8: Variations in Reliability Responses within the `445.gobmk` Benchmark

hybrid, and detection and recovery techniques. This variety is partly due to the diversity of reliability of the native application. The benchmark `435.gromacs`, for example, has a high native reliability with 95.14% of bits labeled unACE and the recovery technique is able to increase it to 98.46% (3.32% increase). `adpcmenc` realizes a larger benefit, with 73.46% of bits labeled unACE before the code transformation to 97.09% after (23.63% increase).

Similar to the variations in the native reliability, the reliability responses also vary within an application. Figures 7.5, 7.6, 7.7, and 7.8 show the breakdown of application reliability by region both before and after the SWIFTR technique is applied to `jpegdec`, `300.twolf`, `433.milc`, and `445.gobmk`, respectively. Likewise, there are regions within an application that greatly deviate from the benchmark average. The benchmark average percentage of unACE bits before SWIFTR is 78.49% and increases to 97.66%, but within the application, certain regions respond more favorably (39.60% change from 56.77% to 96.37%) and others respond less favorably (2.53% change from 94.97% to 97.50%) to the SWIFTR transformation. This reliability response should be considered when deciding what components to protect, but often is not a factor in such decisions. The variety in the reliability response helps determine the regions which would benefit most from the particular technique. SMFT uses this reliability response when determining what is most efficient to protect.

7.3 Variations in Performance Responses

Section 7.2 explored the variations in reliability responses both within and across applications. Similar variations are observed in the cost of the reliability techniques. Execution time after the technique has been applied relative to the unprotected application is referred to as the *performance response*. Figure 6.2 shows the performance response for the SWIFTR technique across the range of applications. The varying performance responses

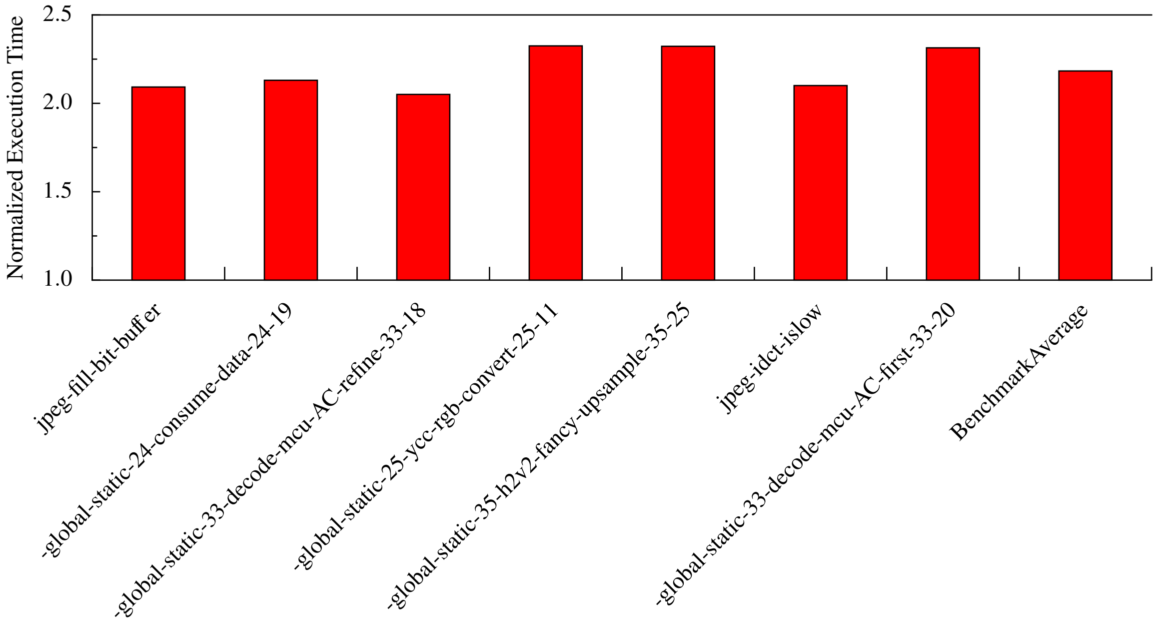

Figure 7.9: Variations in Performance Responses within the `jpegdec` Benchmark

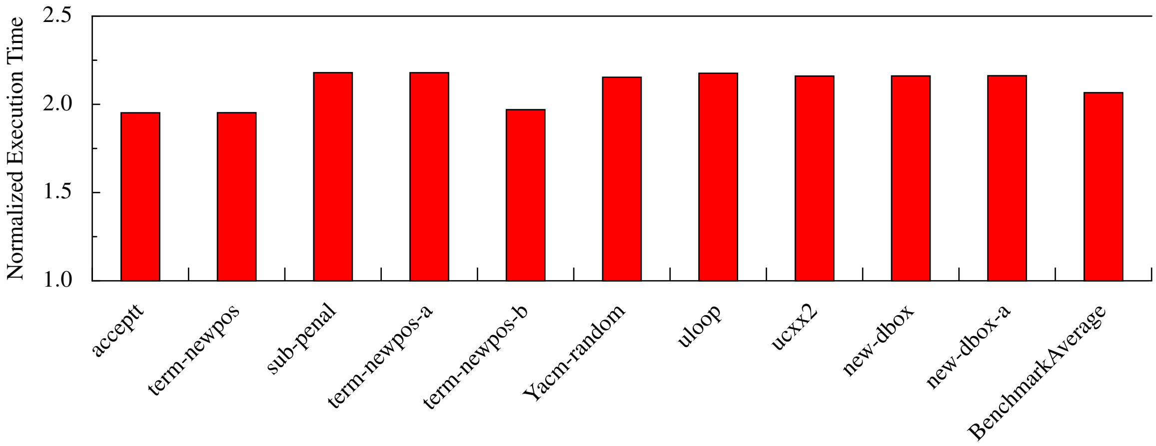

Figure 7.10: Variations in Performance Responses within the `300.twolf` Benchmark

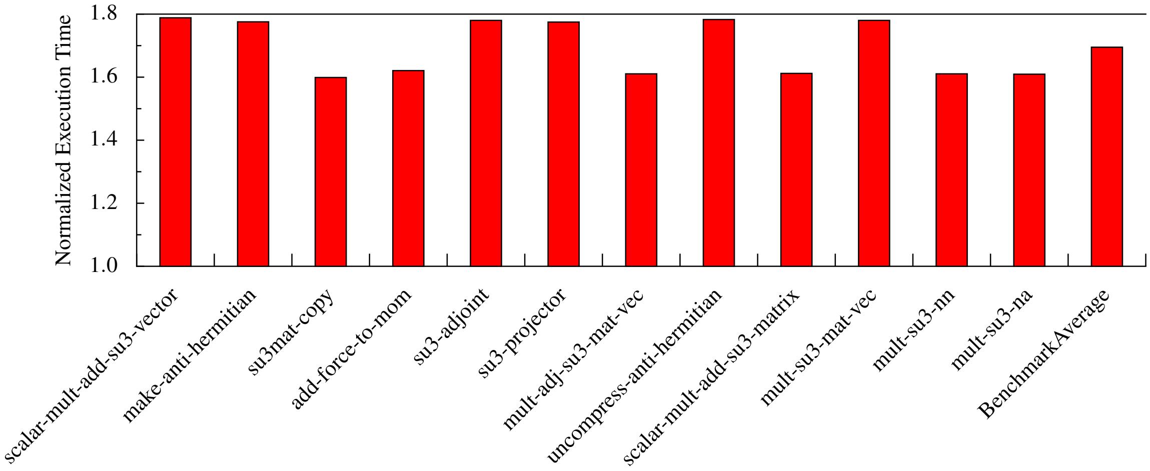

Figure 7.11: Variations in Performance Responses within the `433.milc` Benchmark

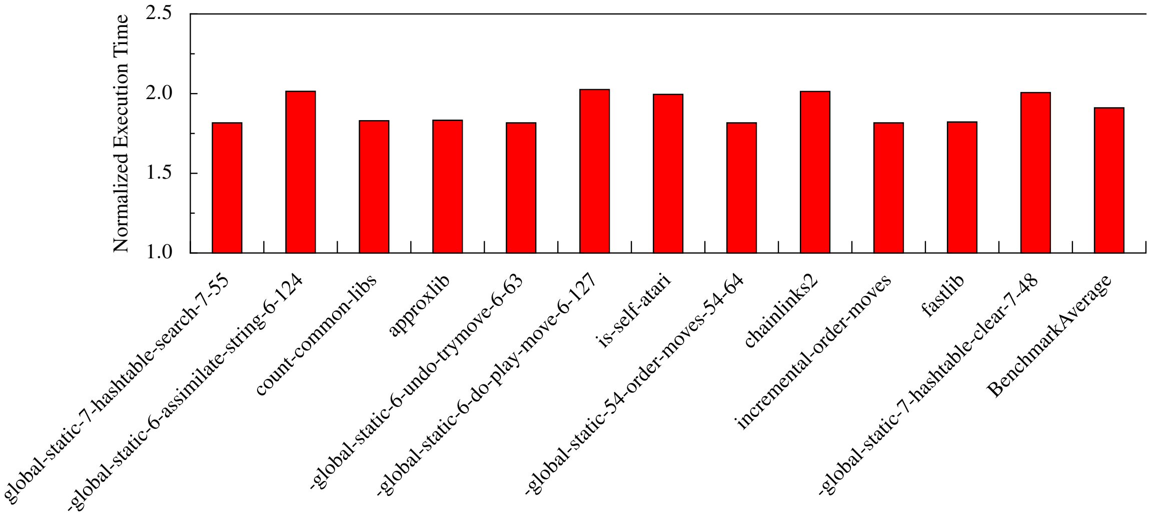

Figure 7.12: Variations in Performance Responses within the `445.gobmk` Benchmark

of SWIFTR range from 1.19x for `429.mcf` to 2.80x for `435.gromacs`. Efficient deployment of the reliability techniques depends upon knowledge of the magnitude of the performance cost. s in the case of the reliability response, the performance response varies for different applications as well as for different regions within an application. Figures 7.9, 7.10, 7.11, and 7.12 show the variety of inter-benchmark performance responses for `jpegdec`, `300.twolf`, `433.milc`, and `445.gobmk`, respectively. The `jpegdec` benchmark has an average normalized execution time of 2.18x but regions within the application range from 2.05x to 2.32x.

The variations in native reliability, reliability response, and performance response lead to large inefficiencies if all applications or regions are treated with uniform protection. SMFT takes advantage of these variations to apply reliability techniques where they will be most effective, thus reducing the overall cost of increasing reliability.

7.4 Choosing Optimally Configured Protection for Specific Constraints

SWIFT and CRAFT, while greatly reducing the number of undetected errors, may also be controlled by software. This allows for precisely management of the level of performance and reliability within a program. An instance of such a technique is SMFT, an algorithm which uses a program's reliability profile to fine-tune the tradeoff between protection and performance. SMFT exploits the fact that regions of code differ intrinsically in terms of reliability. Regions also differ in their natural robustness against transient faults and in their response to various fault-detection schemes. For example, a region that computes a large amount of dynamically dead or logically masked data will naturally mask many transient faults and thus may not require much protection. On the other hand, a region that is dominated by control flow and contains few dynamic instances of store instructions may not need any protection for stores but may instead require large amounts of control-flow

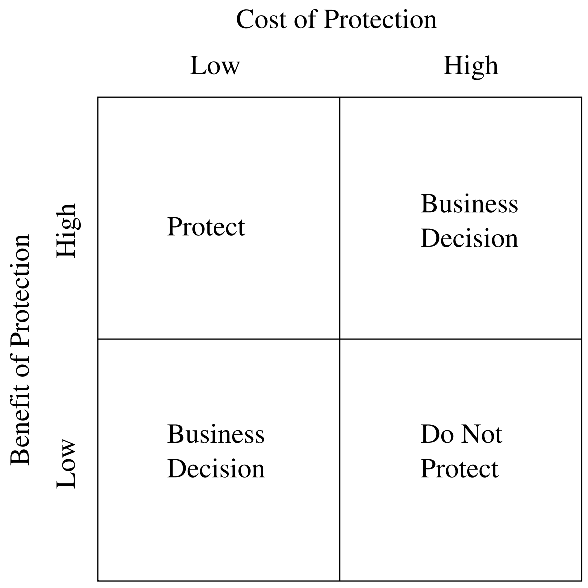

Figure 7.13: Precise Tradeoffs between Reliability and Performance with SMFT

checking.

In general, programs will exhibit a wide range of reliability behaviors. A one-size-fits-all application of any given reliability technique will either be inefficient and over-protect some regions, leave some regions unnecessarily vulnerable, or both. Customizing the protection for each region to maximize the performance and reliability tradeoff is the goal of the SMFT algorithm. By tailoring the reliability to the particular qualities of the region in question, SMFT can obtain reliability comparable to any of the aforementioned protection techniques while simultaneously improving performance.

Figure 7.13 shows four states in which a given region of configurability may exist. The x-axis shows two different costs in term of protection and the y-axis shows two different benefits received from the given reliability transformation. Although real systems have more axes and more options per axis, this simple example illustrates the benefit of SMFT. Previously, with macro-redundancy techniques, all four boxes were treated similarly, as all regions were either simultaneously protected or unprotected. The power of modulated fault tolerance is that it allows precise configurability for each of the boxes. Clearly, the low-cost, high-benefit option (upper left) is one that should be protected and clearly the

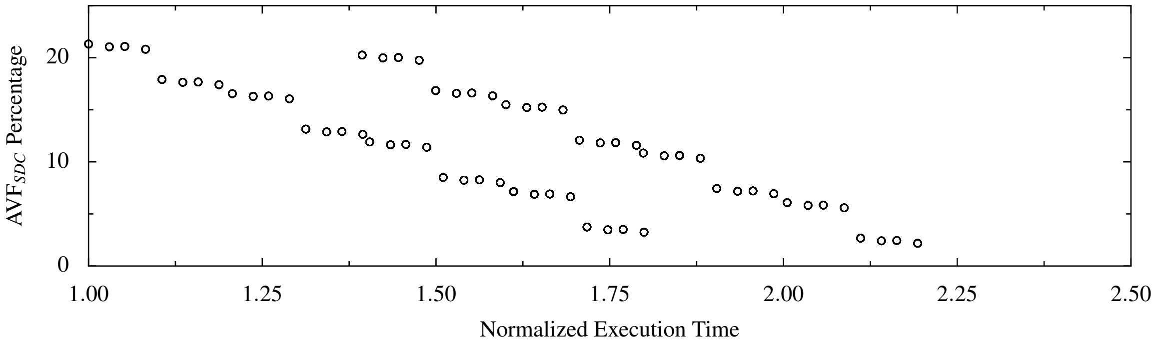

Figure 7.14: Reliability and Performance of Possible Protection Configurations for the `jpegdec` Benchmark

high-cost, low-benefit option (lower right) is one that should not be protected. The other two options are more ambiguous, thus decisions for those cases rely on user preferences.

SMFT operates under the supposition of the existence of a user-defined *utility function*. This utility function should serve as a metric for how desirable a particular version of a program is. Chapter 8 illustrates the reliability and performance for three classes of utility functions: performance constraints, reliability constraints, and efficiency maximization. However, SMFT can be easily extended to utility functions related to other factors such as static size and power.

SMFT can enable tradeoffs between reliability and performance at fine granularities thus maximizing reliability while minimizing the costs. As reviewed earlier, the SWIFT, SWIFTR, CRAFT, and CRAFTR techniques tend to follow the same trends. The two hybrid techniques show similar types of benefits over their respective software-only techniques, and the two recovery techniques offer similar benefits over their respective detection-only techniques. In the interest of clarity and detail, this dissertation focuses on the SWIFTR technique when analyzing details about the configurability options. All of the techniques benefit from SMFT, but the SWIFTR technique has the lowest barrier to use in that there is no hardware requirement while creating the highest percentage of unACE bits.

Figures 7.14, 7.15, and 7.16 show a range of configurations for three different benchmarks, `jpegdec`, `300.twolf`, and `255.vortex`, respectively. These figures show the

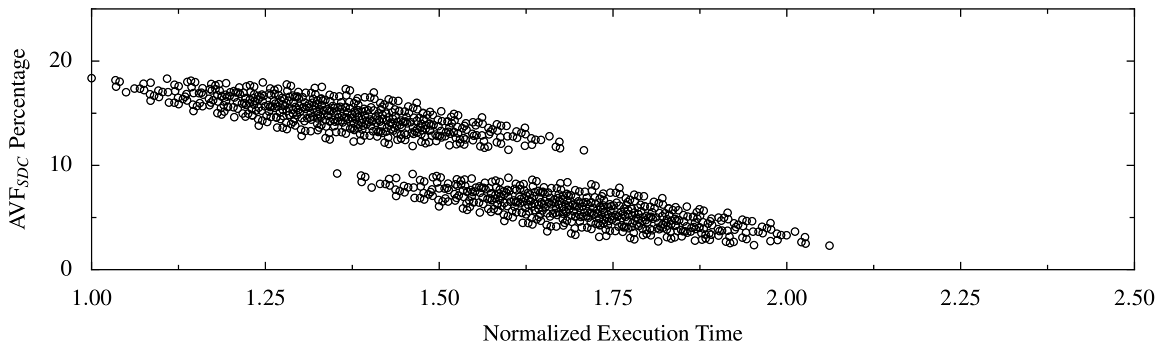

Figure 7.15: Reliability and Performance of Possible Protection Configurations for the `300.twolf` Benchmark

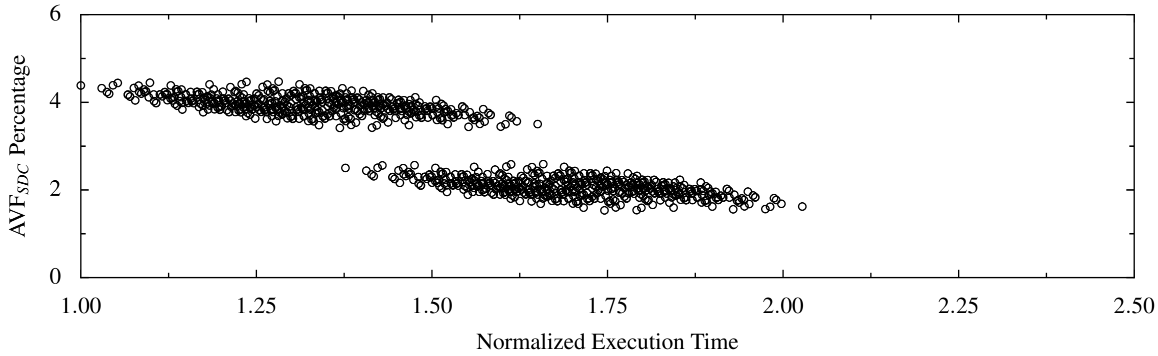

Figure 7.16: Reliability and Performance of Possible Protection Configurations for the `255.vortex` Benchmark

normalized execution time on the x-axis and the AVF$_{SDC}$ for a variety of reliability configurations on the y-axis for each benchmark. Notice that the left side of the x-axis is 1.00, indicating the performance of the unprotected application. The right side of the x-axis is the most expensive in term of performance and represents the full coverage SWIFTR approach. The configuration with a 1.00 normalized execution time and the largest AVF$_{SDC}$ is the unprotected application, on the y-axis near the top of the figure. The configuration with the highest normalized execution time and the lowest SDC is the full protection SWIFTR technique, at the right edge of the x-axis. The points in the middle represent other decisions for protecting verses leaving unprotected particular regions. The optimal position to be on these figures is at the lower left corner, a configuration that has no additional performance cost and an AVF$_{SDC}$ of 0.00% (full reliability). This configuration is often unattainable.

Notice that in Figure 7.14, the points form clusters and follow two basic lines. Clustering is also very noticeable in Figures 7.15 and 7.16. In the last two figures, the clusters correspond to the decisions of whether or not to protect a particular region. This region in each of the benchmarks is important in terms of reliability, but also has a noticeable protection cost. The configurations in the upper left cluster do not protect this region, thus enabling better performance (left side of the figure) at the cost of greater AVF$_{SDC}$ (higher on the figure). The configurations in the lower right all protect this region. Although the scales for Figures 7.15 and 7.16 are different, with `255.vortex` having a smaller and more reliable range for reliability, these benchmarks have the same pattern. This pattern is also present in most of the benchmarks.

The clustering in Figure 7.14 is slightly different than the other benchmarks. There are many four-point clusters that span a small vertical region but a large horizontal region. These four points represent the two configurations (protected or unprotected) for two regions. The small vertical change means that protecting the region does not provide much benefit, but the larger horizontal spread means there is a noticeable performance cost. Leaving these two regions unprotected is almost always beneficial. Also notice the pattern of

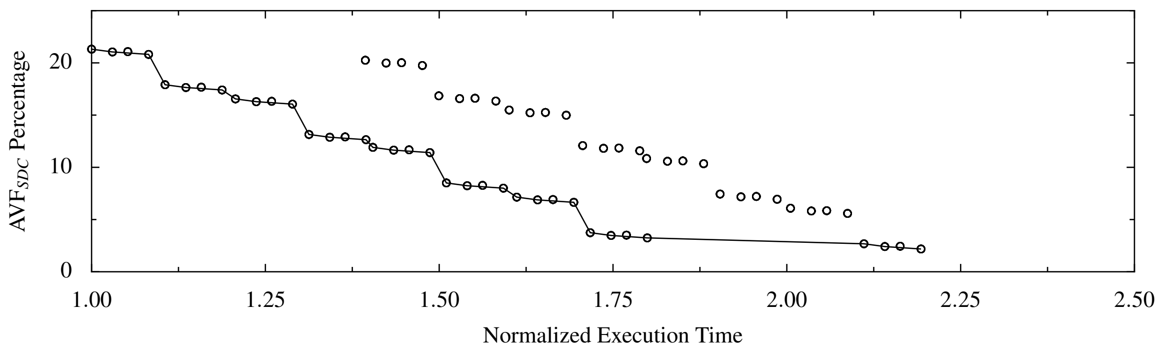

Figure 7.17: Reliability and Performance for the Optimal Set of Configurations for the `jpegdec` Benchmark

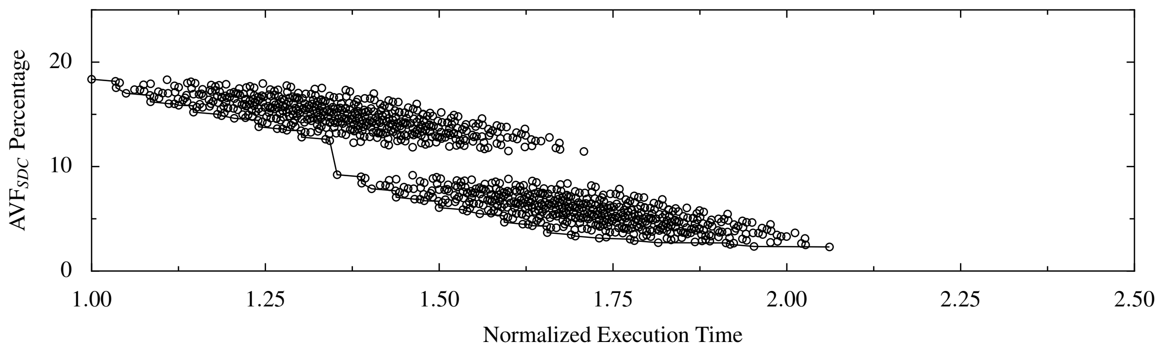

Figure 7.18: Reliability and Performance for the Optimal Set of Configurations for the `300.twolf` Benchmark

two configuration frontiers, which again represents the choice of protecting or leaving unprotected one particular region. This region moves the configuration a significant amount in term of performance cost but not much in terms of reliability and again, in most cases, should be left unprotected.

Although all the points in the performance-reliability scatter plot figures are valid configurations, there are a set of configurations which should never be chosen. That is, a frontier line of configurations illustrates the optimal tradeoff between performance and reliability, and only configurations on that frontier should be chosen. The particular configuration on that frontier is still a function of the constraints and utility specified by the user. Figures 7.17, 7.18, and 7.19 show all of the configurations as well as the frontier line. The frontier line encompasses the configurations that exhibit the lowest AVF$_{SDC}$ for any performance cost and the lowest performance cost for any reliability. They are on the

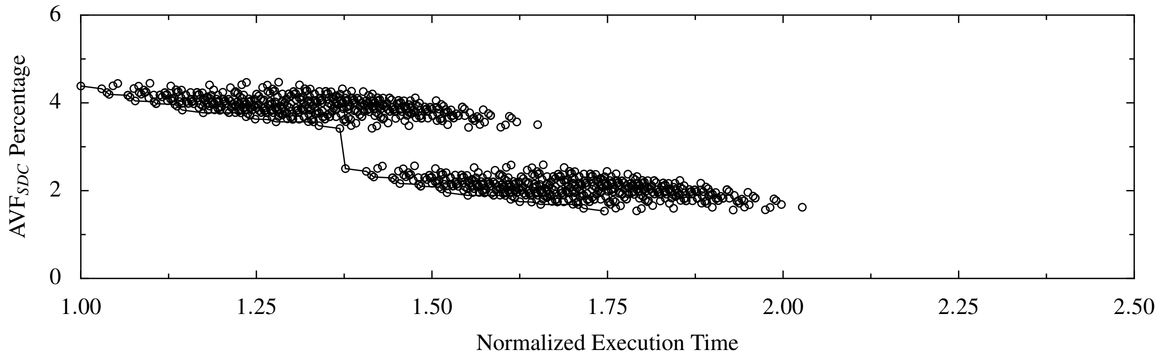

Figure 7.19: Reliability and Performance for the Optimal Set of Configurations for the `255.vortex` Benchmark

frontier of the lower left corner (the high reliability and low cost point). The subsequent chapter of this dissertation will examine three constraints and explore the configurations for the full range of benchmarks. The specific constraints determine which configuration on the frontier line should be selected.

7.5 Relation to Prior Work

Researchers have proposed techniques that modify the RMT implementation to trade reliability for performance [19], while others have proposed techniques that use a combination of prediction with redundant execution to detect transient faults. These techniques use deviations from highly predictable results to trigger redundant execution [49, 42]. Other options for increasing reliability with low overhead include data scrubbing or periodic data flushing in certain hardware structures [76, 6]. Data scrubbing forces any error that can be corrected, via ECC for example, to be corrected and prevents single errors from accumulating to a point where they can no longer be handled. Periodic flushing removes data from hardware structures that have been unused for too long to eliminate possible faults in the data from propagating to the rest of the system. This can be done with little overhead if the flushed data can be easily re-computed. Previous work has explored protecting a subset of the architectural registers [28, 78, 12] or protecting just a subset of data in a protected cache [30, 26].

While researchers are exploring the tradeoff between performance and reliability, most look for large performance gains with minimal reliability reductions. This dissertation is the first exploration of reliability/performance tradeoffs associated with varying the level of protection and finding the most efficient means of protection. Previous techniques do not consider the vulnerability of the unprotected application or the precise reliability and performance responses to the reliability techniques. As reliability is becoming a larger factor in processor development, there will be an increased focus on efficient techniques that can be easily configured to balance reliability and performance considerations.

Chapter 8

Evaluation of Configurable Protection

This chapter describes the quantitative evaluation of the SMFT configurable protection technique by discussing three separate configurations. Section 8.1 illustrates the configurations when SMFT is set to optimize reliability for a given performance requirement. Section 8.2 shows the decisions when SMFT is set to optimize performance for a reliability requirement. Section 8.3 explains the tradeoffs when neither a performance nor reliability constraint is given, but SMFT is set to optimize the tradeoff between both attributes.

8.1 Configurations with Performance Constraints

This section explicates the SMFT advantages when a specific performance constraint is specified. For this version of the technique, an execution time increase of 50% was specified as the maximum acceptable performance penalty. The configurable protection should maximize the reliability for the given application, but not exceed a 1.5x normalized execution time.

Figure 8.1 illustrates the performance of the configurable technique normalized to the unprotected application. Notice that most of the normalized runtimes are within the performance constraint of the configurable protection. `254.gap` has a normalized execution time of 1.51x despite the profiling set to enable a 1.5x runtime. This is an example of

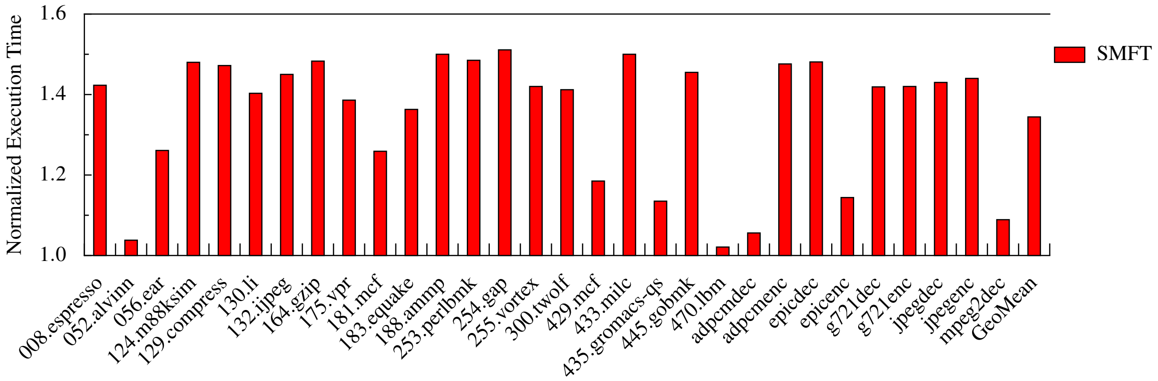

Figure 8.1: Performance of SMFT with Performance Constraint

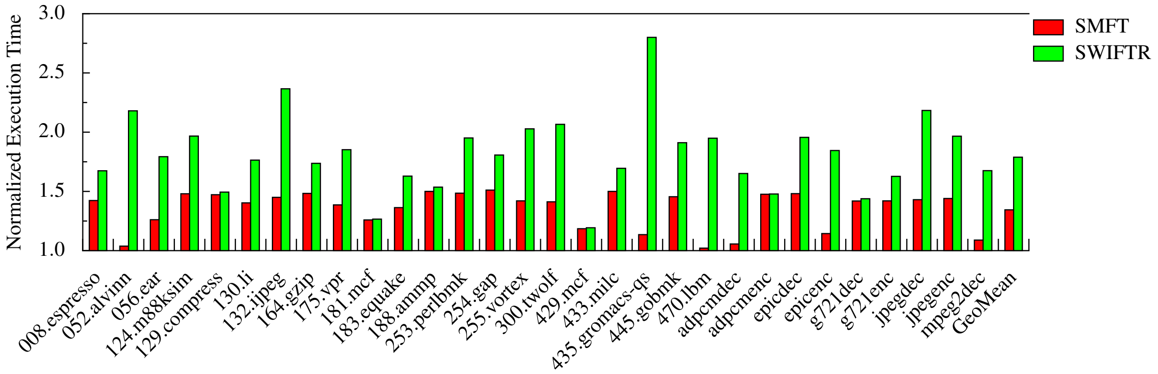

Figure 8.2: Performance of SMFT with Performance Constraint Compared to SWIFTR

system variations slightly altering the final performance. If the 1.5x constraint was a strict requirement, then the configuration for the SMFT technique should be set slightly lower (45% for example) due to the possibility of system performance variations. To optimize performance, this SMFT implementation aims for the exact constraint.

Figure 8.1 shows that some benchmarks, like `056.ear`, `429.mcf`, `470.lbm`, and `adpcmdec`, have normalized execution times that are much less than the 1.5x constraint. Figure 8.2 illustrates the SMFT performance compared with the full protection SWIFTR performance. Notice that for `429.mcf`, both the SMFT and the SWIFTR performance are less than 1.5x. For this benchmark, the SMFT decision is to protect the entire application because even with full protection, the application fits within the performance limit. The same decision is made for `adpcmenc` and `129.compress`, although both of those are closer to the 1.5x limit. If the constraint were set to 1.25x, the `429.mcf` decision would

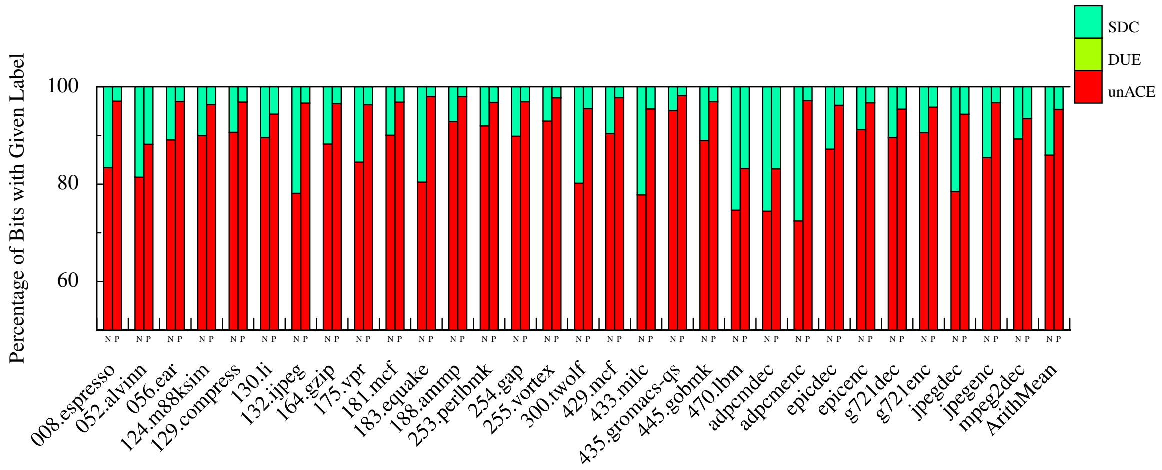

Figure 8.3: Reliability of SMFT with Performance Constraint Compared to NOFT

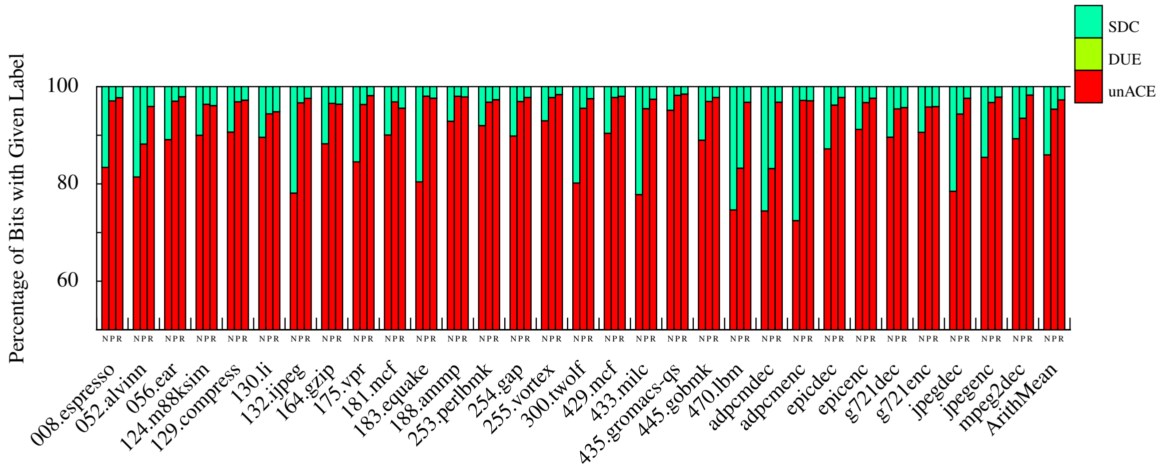

Figure 8.4: Reliability of SMFT with Performance Constraint Compared to NOFT and SWIFTR

be the same, but `adpcmenc` and `129.compress` would have different configurations.

Figure 8.3 shows the reliability of the SMFT configurable technique compared with the unprotected application. For the benchmarks mentioned before that have a very low performance cost, including `056.ear`, `470.lbm`, and `adpcmdec`, the reliability changes leave a significant portion still as AVF_{SDC}. Figure 8.2 illustrates, for these benchmarks, that the cost of enabling full protection is higher than the performance constraint. Full SWIFTR protection for `056.ear`, `470.lbm`, and `adpcmdec` is 2.18x, 1.94x, and 1.65x, respectively.

Figure 8.4 illustrates the configurable technique compared with the SWIFTR technique. For those benchmarks that could not afford full protection, Figure 8.4 shows their reduction in reliability. `470.lbm` has 83.24% of bits labeled as unACE with the SMFT technique, which is between the 74.65% with the unprotected version and the 96.78% with the SWIFTR version. SMFT could be configured to protect more, but that would violate the performance constraint. Benchmarks like `056.ear`, `470.lbm`, and `adpcmdec` have performance costs much lower than 1.5x. These benchmarks were optimized for reliability under the performance constraint, but each have a significant region that is both costly to protect and important for reliability. Protecting this region would have pushed the overall performance cost above 1.5x, hence SMFT did not protect it.

As mentioned before, benchmarks like `adpcmenc` and `129.compress` were able to provide full SWIFTR protection as shown by their performance cost in Figure 8.2. Figure 8.4 illustrates that the reliability of the SMFT technique is equivalent to the SWIFTR technique, which is what is expected from applying SWIFTR to the entire application.

Other benchmarks, like `188.ammp` and `253.perlbmk`, have a performance cost of very near the constraint (1.51x and 1.48x, respectively). These benchmarks were still able able to provide protection for the critical regions of the application, as can be seen in Figure 8.4. The percentage of unACE bits for these two benchmarks is very close to the the percentage of full protection.

SMFT is able to generate protection configurations that obey a specific performance constraint and maximize the reliability within that constraint. For some benchmarks, the configurable reliability approaches that of the full protection technique.

8.2 Configurations with Reliability Constraints

This section explores the SMFT technique when a specific reliability constraint is provided rather than a performance constraint. Figure 8.5 shows this SMFT performance normalized

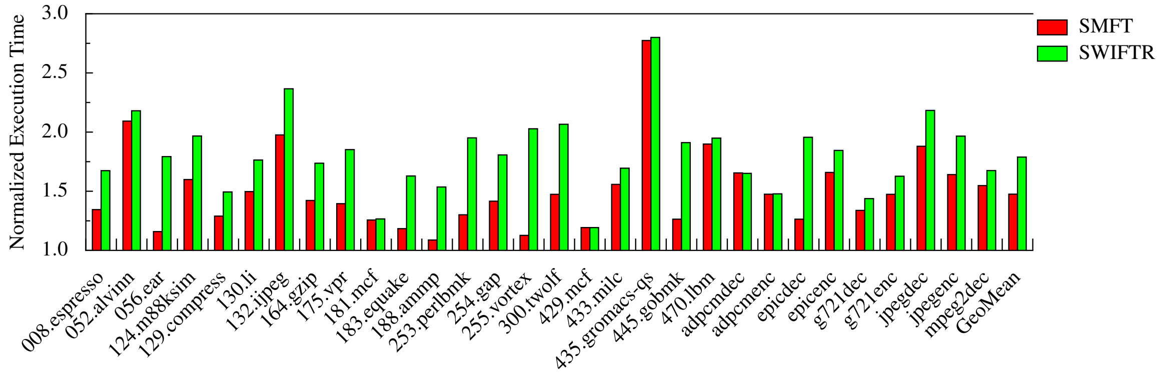

Figure 8.5: Performance of SMFT with Reliability Constraint Compared to SWIFTR

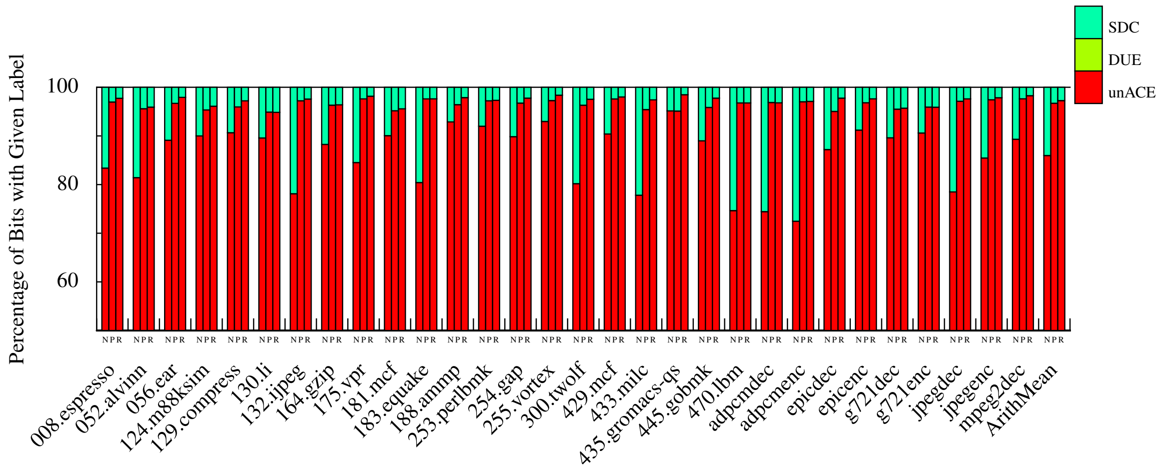

Figure 8.6: Reliability of SMFT with Reliability Constraint Compared to NOFT and SWIFTR

to the unprotected application in addition to the SWIFTR performance for comparison. Figure 8.6 illustrates the reliability of this SMFT technique as well as the unprotected and SWIFTR versions for comparison.

SMFT was able to find a configuration that fit within the 5% AVF$_{SDC}$ limit for each benchmark. Some benchmarks, like `epicdec` with 4.97% and `435.gromacs` with 4.89%, have configurations that use practically all of the reliability budget. The `435.gromacs` case is unique because the natural reliability of the application is 4.86%. This is within the limits of the constraint. The configuration for `435.gromacs` does not protect any of the application. This is illustrated by Figure 8.5, which shows a normalized runtime of 1.01x for `435.gromacs`.

The SMFT configuration for `epicdec` has an AVF$_{SDC}$ of 4.97% whereas the SWIFTR technique has an AVF$_{SDC}$ of 2.23%. Since the constraint was 5.00%, `epicdec` is still within the constraint. As can be seen in Figure 8.5, the SMFT technique has better performance than the full SWIFTR technique. SMFT has a normalized runtime of 1.26x while the SWIFTR technique has a normalized runtime of 1.95x. This performance savings is a result of augmenting the coverage to forfeit 2.5% of AVF$_{SDC}$ to save 0.69x normalized runtime in performance.

`433.milc` explicates another clear example of this tradeoff. This benchmark modulated the performance cost from 1.69x for full protection to 1.55x for SMFT protection (a small 0.14x performance savings). `433.milc` is able to find a configuration that gives a modest savings, but is still able to provide protection. The SMFT configuration is within the AVF$_{SDC}$ constraint with 4.58%, but this is worse than the 2.6% AVF$_{SDC}$ of the SWIFTR technique. SMFT is able to find the configurations that minimize execution time while still remaining within the reliability budget, thus efficiently protecting the application while conforming to the user specifications.

8.3 Optimizing the Tradeoff between Protection and Reliability

This section describes the modulations made to reliability protection when the SMFT technique was not given a specific reliability or performance constraint, but was tasked with finding the most efficient protection for each application.

Figures 8.7 and 8.8 show the performance and reliability of the globally optimizing SMFT technique compared with the software-only recovery technique. Notice in Figure 8.7 that the performance of SMFT is sometimes close to SWIFT and sometimes close to the unprotected application. For `429.mcf`, the SMFT performance is closely aligned with the SWIFTR performance. This is not surprising as the SWIFTR performance cost is low and

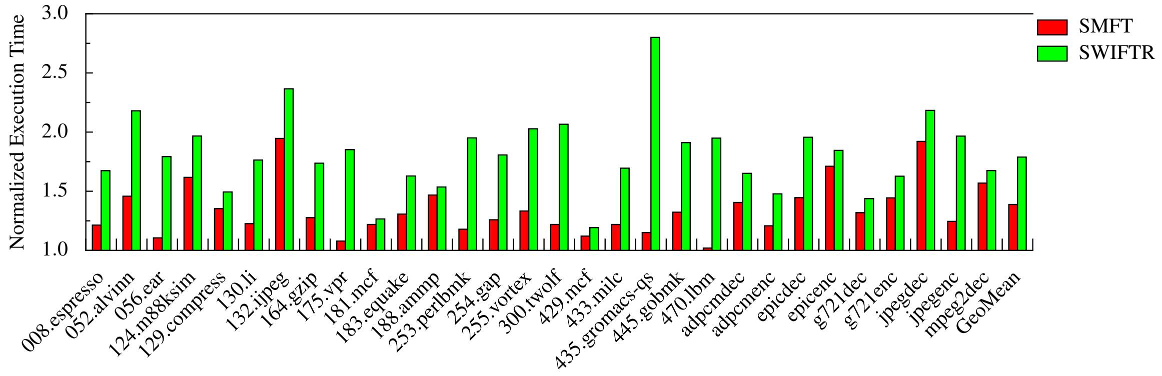

Figure 8.7: Performance of Optimized SMFT Compared to SWIFTR

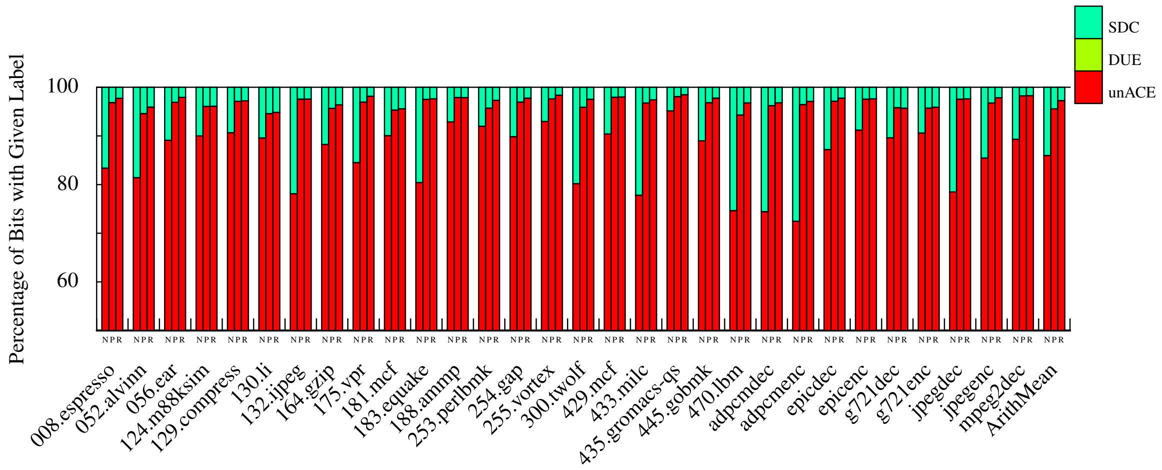

Figure 8.8: Reliability of Optimized SMFT Compared to NOFT and SWIFTR

the reliability benefit is high. The SMFT technique also follows the SWIFTR technique for other costly benchmarks like `epicenc` and `jpegdec`. The SMFT performance cost is high for these techniques, at 1.71x and 1.92x, respectively. For other benchmarks like `130.li`, `435.gromacs`, and `470.lbm`, the performance deviates from the SWIFTR technique.

The reliability consequences of the optimizing technique can be seen in Figure 8.8. As expected, for the benchmarks where performance closely mirrors the SWIFTR technique, the reliability is also similar. For `429.mcf`, the AVF_{SDC} for SMFT is 2.05% and for the full SWIFTR reliability is 1.98%. For some benchmarks that do not closely follow the performance cost of SWIFTR, like `470.lbm`, the reliability benefit is less than

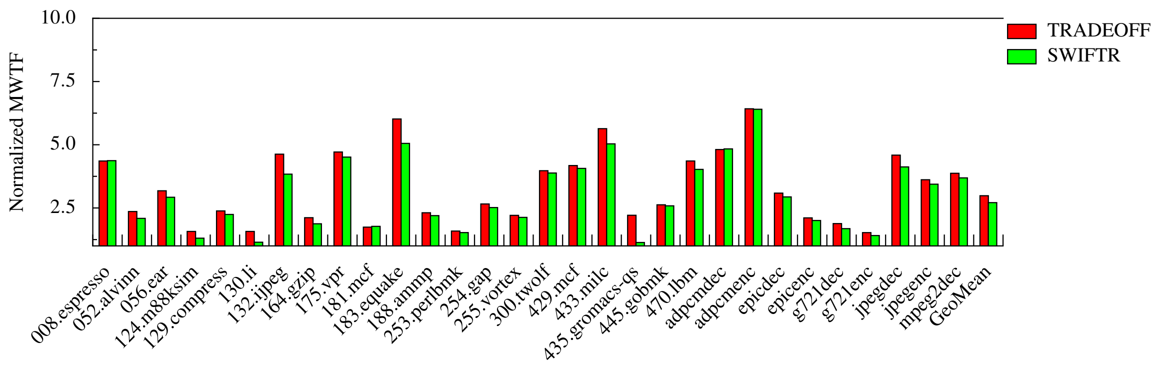

Figure 8.9: Normalized MWTF of Optimized SMFT Compared to SWIFTR

that of SWIFTR. The SMFT AVF$_{SDC}$ is 5.70% while the SWIFTR AVF$_{SDC}$ is 3.23%. These benchmarks make a precise tradeoff between performance and reliability. Although `435.gromacs` does not mirror the performance cost of SWIFTR, the SMFT reliability is much closer to the full protection. Recall that `435.gromacs` is inherently more reliable than most benchmarks. The SMFT AVF$_{SDC}$ is 1.91%, which is close to the SWIFTR AVF$_{SDC}$ of 1.53%.

Figures 8.7 and 8.8 show the performance and reliability evaluation of the configurable technique and for some benchmarks, the SMFT configuration closely follows the SWIFTR configuration while for others it followed the unprotected configuration. This unconstrained SMFT configuration was optimizing for efficient reliability, and specifically MWTF. Figure 8.9 illustrates the MWTF for the SMFT technique normalized to the unprotected application, as well as the nMWTF for the full protection SWIFTR configuration. The x-axis is positioned at an nMWTF value of 1.0, representing the unprotected application.

The SMFT configuration is able to increase the nMWTF of the benchmark suite from 2.99x, up from 2.71x for the SWIFTR technique, by removing protection for the regions that cost too much in terms of performance without providing a corresponding increase in reliability. For some benchmarks, like `adpcmenc` and `adpcmdec`, the modulated technique is only able to find a configuration that is slightly (0.02x) better than the SWIFTR technique. For other benchmarks, like `183.equake` and `132.ijpeg`, SMFT is able to find a configuration that noticeably increases the MWTF. `183.equake` increases from 5.05x to 6.01x and `132.ijpeg` increases from 3.83x to 4.62x. `130.li` and `435.gromacs` have a SWIFTR nMWTF of only 0.13x and 0.14x, respectively, compared to the unprotected application. As reviewed in Section 6.2.2, the SWIFTR technique provides very efficient protection for these benchmarks. The SMFT configuration is able to increase the nMWTF to 1.57x and 2.21x for `130.li` and `435.gromacs`, respectively, demonstrating a clear benefit over the unprotected application.

When free to tradeoff performance and reliability, SMFT is able to optimize for protection efficiency. This SMFT configuration can have a range of performance and reliability responses, but the efficiency, as measured but the nMWTF, is never reduced.

Chapter 9

Future Directions and Conclusions

This dissertation introduced reliability techniques that mitigate the effects of transient faults and enable configurable protection, along with methods to use precise configuration to provide efficient increases in reliability. This chapter discusses potential avenues of future research in Section 9.1 and conclusions based on the work in this dissertation in Section 9.2.

9.1 Future Directions

Potential future research directions based on the contributions of this dissertation stem from both the static and configurable methods of protection. The software-only and hybrid techniques proposed in this dissertation are all compiled without the need for multiple threads of execution. These reliability techniques can be adapted to multi-threaded and multi-core architectures. Related work has been published that extends hardware reliability techniques in this manner [32, 18] and work has already been conducted for software-only techniques [73]. This move to multi-threaded protection can take advantage of the growing trend of manufacturing multiple processor cores per chip. Also, software-only and hybrid implementations can take advantage of traditional type theory to formally prove which values are protected for a given fault model. Preliminary research in this area has already been conducted and has proven the reliability properties of a hybrid detection implementation similar to CRAFT [43].

In the future, it may be possible to measure or predict the reliability of a processor in real time. This would allow for dynamic changes to the reliability requirements and corresponding redundancy levels of the system needed to keep the application and processor executing correctly. Recent research has attempted to predict the AVF from easier-to-measure metrics such as IPC [16, 34, 72]. When combined with modulated fault tolerance, these techniques can be used to dynamically determine which configurations should be used to minimize execution time while still respecting the reliability requirements.

9.2 Conclusions

As transient faults become more prevalent, techniques which can provide the most protection within the reliability budget will become a critical requirement. Finding efficient techniques will support the trend of increased computational power with each processor generation. To this end, this dissertation introduced four techniques which enable transient fault protection with little to no hardware additions. It also introduced a method for configuring those techniques to provide efficient protection tailored to the reliability or performance constraints of the system.

This dissertation presented four transient fault protection techniques that have minor or nonexistent hardware requirements. The software-only fault protection techniques reduce the AVF_{SDC} from 13.56% to 2.39% and 2.74% for the detection-only SWIFT and recovery SWIFTR implementations, respectively. The advantages of the SWIFTR recovery technique are illustrated via the the percentage of bits labeled as unACE. SWIFTR increases the percentage of bits that do not cause program termination from 86.43% to 97.26%. These techniques are also advantageous as they can be easily deployed to existing systems. Although traditionally software-only techniques have been expensive to implement, SWIFT and SWIFTR increase the execution time, on average, by only 54% and 78%, respectively.

This dissertation also introduced two hybrid fault protection techniques, CRAFT and CRAFTR. These techniques require small hardware additions and thus lose some of the benefit of deployment of the software techniques, but they provide higher reliability and improved performance. The CRAFT hybrid detection technique increases the execution time by 43% and reduces the AVF_{SDC} to 1.75%. The CRAFTR hybrid recovery technique increases the execution time by 68% and reduces the AVF_{SDC} to 2.23%. The CRAFTR technique increases the percentage of bits that do not cause program termination to 97.76%.

These techniques, in addition to providing alternatives to traditional hardware-only reliability techniques, can be precisely tuned via the configurable protection technique proposed in this dissertation, SMFT. SMFT provides maximally efficient protection, as it can be used to protect only those regions that can efficiently apply the reliability transformation, thereby saving unnecessary performance losses. This dissertation illustrated how, when configured for efficiency, SMFT can increase the nMWTF by 28%. When configured for a specific execution time constraint of 1.5x, SMFT is able to provide 98% of the reliability of full protection. When SMFT is configured to only allow a 5% AVF_{SDC}, it is able to optimize for that constraint with only a 0.31x reduction in normalized execution time.

This dissertation introduced a range of fault protection techniques, from software-only techniques that provide protection and ease of deployment to hybrid techniques that provide improved protection and performance at the cost of minor hardware additions. This dissertation presented both detection-only and recovery implementation for the software-only and hybrid techniques. Finally, this dissertation introduced the first technique with the ability to precisely configure the reliability and performance of a program at a fine granularity using information about the native reliability of the application in conjunction with the performance and reliability responses of the reliability technique. Such configurable protection allows for efficient reliability, which will be necessary as fault tolerance becomes a first order design concern for computer architects.

Bibliography

[1] *JEDEC Standard: Measurement and Reporting of Alpha Particle and Terrestrial Cosmic Ray-Induced Soft Errors in Semiconductor Devices*. JEDEC Solid State Technology Association, October 2006.

[2] H. Ando, Y. Yoshida, A. Inoue, I. Sugiyama, T. Asakawa, K. Morita, T. Muta, T. Motokurumada, S. Okada, H. Yamashita, Y. Satsukawa, A. Konmoto, R. Yamashita, and H. Sugiyama. A 1.3GHz fifth generation SPARC64 Microprocessor. In *Proceedings of the International Solid-State Circuits Conference*, 2003.

[3] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan. The sorcerer's apprentice guide to fault attacks. *Proceedings of the IEEE*, volume 94, pages 370–382, 2006.

[4] R. C. Baumann. Soft errors in advanced semiconductor devices-part I: the three radiation sources. *IEEE Transactions on Device and Materials Reliability*, volume 1, pages 17–22, March 2001.

[5] R. C. Baumann. Soft errors in commercial semiconductor technology: Overview and scaling trends. *IEEE Reliability Physics Tutorial Notes, Reliability Fundamentals*, pages 121_01.1 – 121_01.14, April 2002.

[6] A. Biswas, P. Racunas, R. Cheveresan, J. Emer, S. S. Mukherjee, and R. Rangan. Computing architectural vulnerability factors for address-based structures. In *Pro-*

ceedings of the 32nd Annual International Symposium on Computer Architecture, 2005.

[7] J. A. Blome, S. Gupta, S. Feng, and S. Mahlke. Cost-efficient soft error protection for embedded microprocessors. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, October 2006.

[8] C. Bolchini and F. Salice. A software methodology for detecting hardware faults in VLIW data paths. In *Proceedings of the IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, 2001.

[9] D. Boneh, R. A. DeMillo, and R. J. Lipton. On the importance of checking cryptographic protocols for faults. *Journal of Cryptology*, volume 14, pages 101–119, 2001.

[10] E. Borin, C. Wang, Y. Wu, and G. Araujo. Software-based transparent and comprehensive control-flow error detection. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 333–345, March 2006.

[11] D. C. Bossen. CMOS soft errors and server design. *IEEE 2002 Reliability Physics Tutorial Notes, Reliability Fundamentals*, pages 121_07.1 – 121_07.6, April 2002.

[12] J. Chang, G. A. Reis, N. Vachharajani, R. Rangan, and D. I. August. Non-uniform fault tolerance. In *Proceedings of the 2nd Workshop on Architectural Reliability*, December 2006.

[13] G. Chen and M. Kandemir. Improving Java virtual machine reliability for memory-constrained embedded systems. In *Proceedings of the 42nd Annual Design Automation Conference*, pages 690–695, 2005.

[14] S. Crosby and J. Melnick. Protecting virtual machines: the "Best of VMworld" approach. Marathon Technologies, April 2007.

[15] E. Fetzer, L. Wang, and J. Jones. The multi-threaded, parity-protected 128-word register files on a dual-core Itanium-family processor. In *Proceedings of the IEEE Internationsl Solid-State Circuits Conference*, volume 1, pages 382–605, Feb 2005.

[16] X. Fu, J. Poe, T. Li, and J. A. B. Fortes. Characterizing microarchitecture soft error vulnerability phase behavior. In *Proceedings of the 14th IEEE International Symposium on Modeling, Analysis, and Simulation*, 2006.

[17] C. Gniady and B. Falsafi. Speculative sequential consistency with little custom storage. In *Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques*, pages 179–188, 2002.

[18] M. Gomaa, C. Scarbrough, T. N. Vijaykumar, and I. Pomeranz. Transient-fault recovery for chip multiprocessors. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 98–109, 2003.

[19] M. A. Gomaa and T. N. Vijaykumar. Opportunistic transient-fault detection. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 172–183, 2005.

[20] S. Govindavajhala and A. Appel. Using memory errors to attack a virtual machine. In *Proceedings of the Symposium on Security and Privacy*, pages 153–165, May 2003.

[21] S. Hareland, J. Maiz, M. Alavi, K. Mistry, S. Walsta, and C. Dai. Impact of CMOS process scaling and SOI on the soft error rates of logic processes. In *Proceedings of the Symposium on VLSI Technology*, pages 73–74, 2001.

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

[23] T. Karnik, B. Bloechel, K. Soumyanath, V. De, and S. Borkar. Scaling trends of cosmic ray induced soft errors in static latches beyond 0.18. In *Proceedings of the Symposium on VLSI Technology*, pages 61–62, 2001.

[24] S. Kim and A. K. Somani. Soft error sensitivity characterization for microprocessor dependability enhancement strategy. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pages 416–425, September 2002.

[25] C. Lee, M. Potkonjak, and W. Mangione-Smith. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 330–335, December 1997.

[26] K. Lee, A. Shrivastava, I. Issenin, N. Dutt, and N. Venkatasubramanian. Mitigating soft error failures for multimedia applications by selective data protection. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 411–420, 2006.

[27] C. McNairy and R. Bhatia. Montecito: a dual-core, dual-thread Itanium processor. *IEEE Micro*, volume 25, pages 10–20, 2005.

[28] G. Memik, M. Kandemir, and O. Ozturk. Increasing register file immunity to transient errors. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 586–591, 2005.

[29] S. E. Michalak, K. W. Harris, N. W. Hengartner, B. E. Takala, and S. A. Wender. Predicting the number of fatal soft errors in Los Alamos national labratory's ASC Q computer. *IEEE Transactions on Device and Materials Reliability*, volume 5, pages 329–335, September 2005.

[30] P. Montesinos, W. Liu, and J. Torrellas. Shield: Cost-effective soft-error protection for register files. In *Proceedings of the 3rd IBM P=ac^2 Conference*, October 2006.

[31] S. S. Mukherjee. *Architecture Design for Soft Errors*. Morgan Kauffman, February 2008.

[32] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt. Detailed design and evaluation of redundant multithreading alternatives. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 99–110, 2002.

[33] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *Proceedings of the 36th Annual International Symposium on Microarchitecture*, pages 29–40, December 2003.

[34] S. Narayanasamy, A. K. Coskun, and B. Calder. Transient fault prediction based on anomalies in processor events. In *Proceedings of the Conference on Design, Automation and Test in Europe*, 2007.

[35] E. Normand. Single event upset at ground level. *IEEE Transactions on Nuclear Science*, volume 43, pages 2742–2750, Dec 1996.

[36] T. J. O'Gorman, J. M. Ross, A. H. Taber, J. F. Ziegler, H. P. Muhlfeld, I. C. J. Montrose, H. W. Curtis, and J. L. Walsh. Field testing for cosmic ray soft errors in semiconductor memories. *IBM Journal of Research and Development*, pages 41–49, January 1996.

[37] N. Oh and E. J. McCluskey. Low energy error detection technique using procedure call duplication. In *Proceedings of the International Symposium on Dependable Systems and Networks*, 2001.

[38] N. Oh, P. P. Shirvani, and E. J. McCluskey. Control-flow checking by software signatures. *IEEE Transactions on Reliability*, volume 51, pages 111–122, March 2002.

[39] N. Oh, P. P. Shirvani, and E. J. McCluskey. ED^4I: Error detection by diverse data and duplicated instructions. *IEEE Transactions on Computers*, volume 51, pages 180 – 199, February 2002.

[40] N. Oh, P. P. Shirvani, and E. J. McCluskey. Error detection by duplicated instructions in super-scalar processors. *IEEE Transactions on Reliability*, volume 51, pages 63–75, March 2002.

[41] J. Ohlsson and M. Rimen. Implicit signature checking. In *Proceedings of the International Conference on Fault-Tolerant Computing*, June 1995.

[42] A. Parashar, A. Sivasubramaniam, and S. Gurumurthi. SlicK: slice-based locality exploitation for efficient redundant multithreading. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages And Operating Systems*, pages 95–105, 2006.

[43] F. Perry, L. Mackey, G. A. Reis, J. Ligatti, D. I. August, and D. Walker. Fault-tolerant typed assembly language. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2007.

[44] Perfmon: An IA-64 performance analysis tool. http://www.hpl.hp.com/research/linux/perfmon.

[45] R. Phelan. Addressing soft errors in ARM core-based SoC. ARM White Paper, December 2003.

[46] M. Prvulovic, Z. Zhang, and J. Torrellas. ReVive: Cost-effective architectural support for rollback recovery in shared-memory multiprocessors. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, May 2002.

[47] J. Ray, J. C. Hoe, and B. Falsafi. Dual use of superscalar datapath for transient-fault detection and recovery. In *Proceedings of the 34th Annual International Symposium on Microarchitecture*, pages 214–224, 2001.

[48] M. Rebaudengo, M. S. Reorda, M. Violante, and M. Torchiano. A source-to-source compiler for generating dependable software. In *Proceedings of the IEEE International Workshop on Source Code Analysis and Manipulation*, pages 33–42, 2001.

[49] V. K. Reddy, E. Rotenberg, and S. Parthasarathy. Understanding prediction-based partial redundant threading for low-overhead, high- coverage fault tolerance. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 83–94, 2006.

[50] S. K. Reinhardt and S. S. Mukherjee. Transient fault detection via simultaneous multithreading. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 25–36, 2000.

[51] G. A. Reis, J. Chang, and D. I. August. Automatic instruction-level software-only recovery methods. *IEEE Micro Top Picks*, volume 27, January 2007.

[52] G. A. Reis, J. Chang, D. I. August, R. Cohn, and S. S. Mukherjee. Configurable transient fault detection via dynamic binary translation. In *Proceedings of the 2nd Workshop on Architectural Reliability*, December 2006.

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

[54] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, D. I. August, and S. S. Mukherjee. Design and evaluation of hybrid fault-detection systems. In *Proceedings of the 32th Annual International Symposium on Computer Architecture*, pages 148–159, June 2005.

[55] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, D. I. August, and S. S. Mukherjee. Software-controlled fault tolerance. *ACM Transactions on Architecture and Code Optimization*, volume 2, pages 366–396, December 2005.

[56] E. Rotenberg. AR-SMT: A microarchitectural approach to fault tolerance in microprocessors. In *Proceedings of the 29th Annual International Symposium on Fault-Tolerant Computing*, pages 84–91, June 1999.

[57] N. Saxena and E. J. McCluskey. Dependable adaptive computing systems – the ROAR project. In *Proceedings of the International Conference on Systems, Man, and Cybernetics*, pages 2172–2177, October 1998.

[58] M. A. Schuette and J. P. Shen. Exploiting instruction-level parallelism for integrated control-flow monitoring. *IEEE Transactions on Computers*, volume 43, pages 129–133, February 1994.

[59] J. Segura and C. F. Hawkins. *CMOS Electronics: How It Works, How It Fails*. Wiley-IEEE Press, April 2004.

[60] P. P. Shirvani, N. Saxena, and E. J. McCluskey. Software-implemented EDAC protection against SEUs. *IEEE Transactions on Reliability*, volume 49, pages 273–284, 2000.

[61] P. Shivakumar, M. Kistler, S. W. Keckler, D. Burger, and L. Alvisi. Modeling the effect of technology trends on the soft error rate of combinational logic. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 389–399, June 2002.

[62] A. Shye, V. J. Reddi, T. Moseley, and D. A. Connors. Transient fault tolerance via dynamic process redundancy. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, October 2006.

[63] T. J. Slegel, R. M. Averill III, M. A. Check, B. C. Giamei, B. W. Krumm, C. A. Krygowski, W. H. Li, J. S. Liptay, J. D. MacDougall, T. J. McPherson, J. A. Navarro, E. M. Schwarz, K. Shum, and C. F. Webb. IBM's S/390 G5 microprocessor design. *IEEE Micro*, volume 19, pages 12–23, March 1999.

[64] J. C. Smolens, B. T. Gold, J. Kim, B. Falsafi, J. C. Hoe, and A. G. Nowatzyk. Fingerprinting: Bounding soft error detection latency and bandwidth. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 224–234, October 2004.

[65] D. J. Sorin, M. M. K. Martin, M. D. Hill, and D. A. Wood. SafetyNet: Improving the availability of shared memory multiprocessors with global checkpoint/recovery. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, May 2002.

[66] T.C. and M. H. Woods. Alpha-particle-induced soft errors in dynamic memories. *IEEE Transactions on Electron Devices*, volume ED-26, pages 2–9, 1979.

[67] Y. Tosaka, S. Satoh, K. Suzuki, T. Sugii, H. Ehara, G. Woffinden, and S. Wender. Impact of cosmic ray neutron induced soft errors on advanced submicron CMOS circuits. In *Proceedings of the Symposium on VLSI Technology*, pages 148–149, June 1996.

[68] M. Tremblay and Y. Tamir. Support for fault tolerance in VLSI processors. In *Proceedings of the IEEE International Symposium on Circuits and Systems*, volume 1, pages 388–392, May 1989.

[69] S. Triantafyllis, M. J. Bridges, E. Raman, G. Ottoni, and D. I. August. A framework for unrestricted whole-program optimization. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 61–71, June 2006.

[70] R. Venkatasubramanian, J. P. Hayes, and B. T. Murray. Low-cost on-line fault detection using control flow assertions. In *Proceedings of the 9th IEEE International On-Line Testing Symposium*, pages 137–143, July 2003.

[71] T. N. Vijaykumar, I. Pomeranz, and K. Cheng. Transient-fault recovery using simultaneous multithreading. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 87–98, 2002.

[72] K. R. Walcott, G. Humphreys, and S. Gurumurthi. Dynamic prediction of architectural vulnerability from microarchitectural state. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, June 2007.

[73] C. Wang, H. Kim, Y. Wu, and V. Ying. Compiler-managed software-based redundant multi-threading for transient fault detection. In *Proceedings of the International Symposium on Code Generation and Optimization*, March 2007.

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

[75] N. J. Wang, J. Quek, T. M. Rafacz, and S. J. Patel. Characterizing the effects of transient faults on a high-performance processor pipeline. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 61–72, June 2004.

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

[77] J. H. Wensley, M. W. Green, K. N. Levitt, and R. E. Shostak. The design, analysis, and verification of the SIFT fault tolerant system. In *Proceedings of the 2nd International Conference on Software Engineering*, pages 458–469, 1976.

[78] J. Yan and W. Zhang. Compiler-guided register reliability improvement against soft errors. In *Proceedings of the 5th ACM International Conference on Embedded Software*, pages 203–209, 2005.

[79] Y. Yeh. Triple-triple redundant 777 primary flight computer. In *Proceedings of the IEEE Aerospace Applications Conference*, volume 1, pages 293–307, February 1996.

[80] J. F. Ziegler, H. W. Curtis, H. P. Muhlfeld, C. J. Montrose, B. Chin, M. Nicewicz, C. A. Russell, W. Y. Wang, L. B. Freeman, P. Hosier, L. E. LaFave, J. L. Walsh, J. M. Orro, G. J. Unger, J. M. Ross, T. J. O'Gorman, B. Messina, T. D. Sullivan, A. J. Sykes, H. Yourke, T. A. Enger, V. Tolat, T. S. Scott, A. H. Taber, R. J. Sussman, W. A. Klein, and C. W. Wahaus. IBM experiments in soft fails in computer electronics (1978 - 1994). *IBM Journal of Research and Development*, volume 40, pages 3–18, January 1996.

[81] J. F. Ziegler and H. Puchner. *SER - History, Trends, and Challenges: A Guide for Designing with Memory ICs*. Cypress Semiconductor Corporation, 2004.