Understanding and Optimizing Heterogeneous Soft-Error Protection

---

A Dissertation

Presented to

the Faculty of the School of Engineering and Applied Science

University of Virginia

In Partial Fulfillment

of the requirements for the Degree

Doctor of Philosophy (Computer Engineering)

by

Lukasz G. Szafaryn

May 2015

Abstract

The trend of continuing technology scaling in circuits exaggerates effects of physical phenomena, such as particle strikes [1] and process variation [2], that cause soft errors . While recent advances in fabrication technology decrease the severity of these effects for the next transistor generation [3] [4], the trend of the increasing error rates inevitably continues with further scaling or change of the operating environment. As a result, it is not becoming necessary to protect a larger range of processor types, including commodity products, in addition to high-end servers [5] and safety-critical systems [6]. Maintaining the same level of resiliency with the constantly increasing processor complexity requires novel approaches to use a wider range of resiliency techniques for improved coverage and efficiency.

While there is a broad range of existing and proposed protection techniques that span hardware, architecture and software layers, only a few of them made it to commercial products while most have only been analyzed individually on inconsistent platforms. Therefore there is little understanding of relative trade-offs between available solutions [7] which makes it difficult to arrive at optimal decisions for the resilient design. Moreover, none of the previous work attempted to compare the more efficient techniques from architecture and software layers and analyze their combinations with hardware-level solutions that could potentially tailor to the protection needs better. Furthermore, there is little analysis of recovery cost, crucial to the efficiency of protection.

To better understand and optimize soft-error protection, we study the following research issues:: 1) the design of a framework for the analysis of protection overheads of traditionally used resilience solutions at different core or component granularities for the OpenRISC sample processor, 2) implementation of data-flow checking, an efficient architecture-level resilience technique, for the Leon and IVM processors to compare against low-overhead hardware and software level techniques, their complementary combinations and recovery methods, as well as 3) analysis of the hardware algorithm-specific protection technique for the FFT accelerator in comparison to the traditionally used hardware solutions to demonstrate the benefits of the algorithm-specific approach for accelerators.

Approval Sheet

This dissertation is submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy (Computer Engineering)

Lukasz G. Szafaryn

This dissertation has been read and approved by the Examining Committee:

Kevin Skadron, Advisor

Joanne Dugan, Committee Chair

Mircea Stan

James Cohoon

Brett Meyer

Accepted for the School of Engineering and Applied Science:

James H. Aylor, Dean, School of Engineering and Applied Science

May 2015

To everyone who has helped me succeed

Acknowledgments

First, I would like to express my gratitude to my advisor, Prof. Kevin Skadron, for his continuous guidance and support throughout my graduate studies at the University of Virginia. His insights and useful remarks helped me develop my research skills, improve the quality of work and overcome many challenges while pursuing the Ph.D. degree.

My special thanks got to Prof. Mircea Stan for his contributions to the reliability research that became part of this dissertation, as well as for useful comments on the oral and written parts of the proposal and dissertation.

I would also like to thank my committee, Prof. Joanne Bechta Dugan, Prof. Mircea Stan, Prof. Jim Cohoon and Prof. Brett Meyer, for taking time to review my proposal and dissertation as well as for giving me many valuable comments about this work.

Much of my research work was inspired by interesting and challenging projects during my multiple internships. I would like to thank Lawrence Livermore National Laboratories and Intel for providing these opportunities during the course of my studies.

I would like to thank my colleagues from the lava lab: Michael Boyer, Shuai Che, Jiayuan Meng, David Tarjan, Mario Marino, Marisabel Guevara, Liang Wang and Runjie Zhang, and many others for helpful discussions and comments that helped me think through many of the research problems.

Finally and most of all, I would like to thank my mother for her continuous support, understanding and encouragement during my graduate studies.

This work was sponsored by SRC and Freescale under SRC task no. 2042 as well as by DARPA MTO under contract no. HR0011-13-C-0022.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Problem Overview

The increasing computation capability of processors requires continuation of the scaling trend to allow fitting more features onto the die. However, at the same time, the decreasing feature sizes make circuits more susceptible to permanent and transient faults. The former are usually caused by manufacturing defects and circuit aging, while the latter are due to charged particle strikes [1] or voltage droops [2]. In this work, we focus on transient faults, known as soft errors, which are more common and lead to interesting protection trade-offs due to masking at different level of the design stack. Soft errors are becoming more prominent because smaller feature capacitance makes it easier for an induced charge to change the logic state at a given, now decreased, operating voltage. The decreased feature size also exaggerates lithographic imperfections that result in varied timing characteristics across the circuit and increased effects of voltage droops. Furthermore, the growing power consumption in the core necessitates employment of drastic power-saving techniques such as dynamic voltage and frequency scaling (DVFS). These techniques exacerbate the soft-error problem by lowering critical charge, prolonging exposure and potentially causing timing violations [11]. As a result, the fraction of the core devoted to protection against these effects keeps growing to maintain the same level of resiliency.

Recent advances in fabrication technology, such as multi-gate transistors [3] and silicon-on-insulator (SOI) [4], counteract these effects by decreasing area in the transistor that is sensitive to particle strikes, thus noticeably lowering the soft-error vulnerability. However, the trend of the increasing error rates inevitably continues in the next transistor generations with further scaling . In our analysis, we focus on soft errors in logic circuits since there already exists many well-established techniques for protecting regular, array-type

structures like memories. For logic circuits, we look at errors in sequential logic (flip-flops) since combinational logic is shown to be far less susceptible [3]. Among the traditionally used solutions for sequential logic, parity provides only limited protection at low overhead, while coarse-grained replication gives more comprehensive coverage at a prohibitively high overhead [12]. Moreover, the traditional, coarse-grained, determination of vulnerability at the core component level leads to unnecessarily excessive overheads are becoming less tolerable. Therefore maintaining the same level of resiliency with the constantly increasing processor complexity requires novel approaches to consider a wider range of resiliency techniques for improved coverage and lower footprint.

The above challenges with providing resiliency for the next generations of processors suggest the need for more precise ways to determine vulnerable features as well as more fine-grained and flexible protection approaches to achieve desired coverage at lower cost. While there are many existing and proposed resiliency solutions, they have all been studied in isolation on various inconsistent platforms. As a result there is no common baseline for evaluation of these solutions that is needed to understand their relative benefits and to arrive at optimal design choices. Moreover, each of the existing solutions alone, suffers from one or more deficiencies of high area, power or performance overhead. Therefore our work does not focus on creating another resilience technique, but rather tries to illustrate how to achieve optimal resilience at a low cost with existing solutions. For any given design, one should be able to chose from a wider range of protection techniques including software, architecture and hardware, individually or in complementing combinations in order to find the best fit at the level of a core, component or an application. Since this broad design space has not been studied collectively, there is a set of important research questions that this work aims to address in order to understand how to further optimize efficiency of soft error protection.

There is little previous work on illustrating relative trade-offs between traditionally used protection techniques, such as hardening [13] [14] [15], error detecting/correcting codes and replication, in the context of different types of circuits components in the core. It is not obvious how the overheads of these solutions grow with the more comprehensive coverage as well as whether they have an effect on the optimal choice of protection. Among the available low-overhead protection choices, there are no studies showing how architecture-level techniques such as control/data-flow checking [16] [17] compares to fine-grained circuit-level solutions such as selective hardening as well as algorithm-specific software level solutions [18] [10]. It is unclear how much of the circuit can be efficiently covered by each while, depending on the requirements and constraints, requiring complementary coverage from others. Finally, for a specialized platform such as an accelerator, we do not know how the traditional circuit-level protection techniques such as hardening, error detecting/correcting codes and replication compare to an specific algorithm-based solution. While the latter is expected to be a more efficient choice, the efficiency of this checking mechanism, in terms of coverage and area overhead, needs to evaluated against traditional solutions.

1.2 Approach

To address the aforementioned concerns, the dissertation addresses two major research challenges:

1. Understanding the applicability of different protection techniques, according to the types of circuits and components in the core, as well as their relative efficiency in terms of area, performance and general coverage. This involves implementation and comparison of these solutions on common platforms.

2. Finding optimal ways to achieve desired coverage for given design constrains with more efficient techniques or their cross-layer (hardware, architecture, software) combinations applied according to their best fit. This involves analysis of complementary solutions via synthesis and error injection.

The main hypothesis of this work is that the improvement in the efficiency of a resilient design requires consideration of a wider range of more efficient, cross-layer, protection techniques or their combinations as well as more precise ways of determining vulnerability. Such an approach should allow for better tailoring to protection needs of a given system and its constraints. We evaluate this hypothesis by conducting research in Item 2. Research performed in Item 1 is necessary to illustrate the sources of overheads and relative coverage capabilities or various resiliency solutions on the same platforms as well as to illustrate benefits of low-level solutions analyzed in Item 2.

To address the above research challenges, our approach consists of multiple major research items:

1. Illustrating differences among processor components in terms of resiliency requirements as well as understanding sources of relative overheads of various, traditionally used, applicable protection solutions.

2. Analyzing a set of efficient architecture, hardware and software techniques and their cross-layer combinations to optimize core protection while stressing the importance of recovery and accurate vulnerability determination.

3. Demonstrating an opportunity for improved performance and coverage achievable with algorithm-based protection approach in a fixed-function accelerator unit compared to traditional approaches used there.

Each of the previous research items contributes to the overall mission of understanding the resilient design space and optimizing protection choices.

To better understand the scope and overhead of different, traditionally used, protection techniques (hardening, parity, ECC, replication, residue codes), we first implement them for the OpenRISC processor and apply in combinations to relevant components according to circuit type (Item 1) as a part of the Svalinn framework that we develop. This provides per-component relative area overheads and general coverage

estimates that we use to analyze the overhead growth in light of the increased protection needs. We further study this aspect, in collaboration with Stanford University, by considering a set of more competitive low-overhead techniques (hardening, control/data-flow integrity and an algorithm-based approach) for the Leon3 processor to analyze their coverage as well as protection and recovery overheads (Item 2). Moreover, we study benefits of the algorithm-based approach in more detail by implementing it in software and hardware accelerator form for the Fast Fourier Transform (FFT) (Item 3).

To demonstrate how to achieve and optimize comprehensive protection in light of increasing resiliency needs, we first evaluate the coverage cost of the aforementioned traditional resiliency techniques and analyze alternative choices among them. We then present multiple combinations of these solutions that maximize coverage while optimizing for different overhead metrics such as area, delay and power (Item 1). Informed by our study in Item 1, we further pursue this direction by focusing on a set of the aforementioned more efficient techniques that we evaluate individually and in combinations for particular coverage and overhead targets (Item 2). We use these results to analyze how much the overall processor protection can be optimized with a more precise determination of vulnerability at a flip-flop level as well as by applying resiliency solutions according to their best fit. Finally, we evaluate the benefits of an algorithm-based protection approach for the FFT accelerator unit (Item 3).

1.3 Characterization of the Protection Overhead

Although researchers have been aware of the effects of radiation-induced faults in non-memory circuits for a long time, until recently, it has only been a concern for massively parallel, accuracy-sensitive and safety-critical architectures. Because of the low severity of the problem as well as its relevance being constrained to niche markets and premium priced products, only a few of the known protection techniques made it to commercial processors, such as those from IBM [5] and Freescale [6], to address this issue. The resiliency solutions traditionally used in those architectures include hardening [13] [14] [15], parity, error detecting/correcting codes (ECC) and replication [12] as well as parity prediction and residue codes [19] in some cases. These techniques are usually applied selectively according to vulnerability determined at a coarse-grained level of a processor or its components. However, as the soft error problem becomes a more important concern with the increasing complexity of processors, such coarse-grained resiliency solutions may no longer be affordable due to the unnecessarily excessive overhead. Therefore it becomes necessary to better understand fine-grained vulnerabilities in the core to apply optimal protection at that level.

However, there are several important research questions that need to be answered to further understand and explore the existing design space. There is no previous study of the various existing protection techniques

or their combinations in the context of a single processor. Such analysis would be useful in serving as a baseline for evaluation of these solutions as well as a vehicle to improve understanding of the relative sources of overheads in each technique and core component. Since core components have different functionality and include varying proportions of cell types, researchers should determine how the per-component overhead of resiliency would change when a more comprehensive or stronger applicable resiliency solution is used. Moreover, since more processor units are now expected to become vulnerable with the increasing error rates, it is important to analyze the overall cost of protection at the level of a core, which is particularly important for the small throughput-oriented cores. Furthermore, researchers should observe whether the need for more comprehensive protection affects the relative benefits of individual resiliency features and translates to different optimal protection choices. Finally, since the added protection features change the area, power and delay characteristics of the core components, it is be important to show how to optimize the overall core-level resiliency according to these metrics.

In Chapter 2, we implement a set of protection techniques for the OpenRISC example processor core to study their overheads and coverage characteristics at the level of a core and its components. The considered resiliency solutions include hardening, parity, ECC, arithmetic codes (parity prediction and residue) and spatial/temporal redundancy. The source data for the analysis is obtained by synthesizing protection configurations at various granularities in the core. For convenience, the data is placed in an analytical framework, called Svalinn, that is capable of estimating overheads for resilience scenarios consisting of any range of flip-flips, core component or for the entire core. Among many possible analyses, we first present a per-component breakdown to show differences in vulnerability and the cost of coverage. Subsequently, we break the select protection techniques into data and logic to illustrate the contribution of each. Finally, another important outcome of our work is an exploration of interesting combinations of techniques for the comprehensive protection of the core. We apply these solutions according to their best fit to illustrate how the corresponding overall area, power and performance metrics can be improved. To conclude our work, we offer a discussion on the challenges with achieving efficient protection with replication and stronger coding which motivates our work in the subsequent chapter.

1.4 Resilience Optimization with Cross Layer Techniques

Our work in the previous illustrates challenges with the current approach to the resilient design that at related to the high overhead of the commonly used protection techniques and coarse-grained methods of determining vulnerability. While more comprehensive implementations of replication and error codes are too expensive for logic circuits, hardening, parity and arithmetic codes are still viable solutions. Also as a

result, there is a growing need to evaluate these techniques against other, proposed, competitive solutions such as architecture-level data-flow checking (DFC) [16] [17] and algorithm-based fault tolerance (ABFT) [18] [10] that provide efficient coverage for particular functionality in the core. For the more comprehensive protection, these solutions may need to be applied in complementary cross-layer combinations with efficient generic solutions, such as the aforementioned hardening [13] [14] [15] and parity [20], for the protection of the remaining circuits for a given resilience target. The previous work determines DFC to be a competitive technique along with selective hardening, parity and ABFT that also show benefits over traditional solutions. However, in the particular case of the DFC, there is a need to determine how it compares to the remaining techniques within its own scope of protection, and whether it can benefit overall coverage in cross-layer combination with those.

However, there are several important challenges that need to be addressed to optimize resilient design with above cross-layer approach. First, researchers should determine how much of the relevant hardware could be covered by functionality-specific solutions, such as DFC and ABFT. For DFC, this involves coverage of control-flow related features in the processor pipeline, while for ABFT it is the specific data in the application. Subsequently, it should be evaluated whether these specific solutions can provide the most efficient protection for their relative coverage scope compared to generic techniques such as hardening and parity. Finally, researchers should estimate how much of the additional, generic techniques, such as hardening and parity, would still be required for the remaining vulnerable circuitry to reach a given resilience target. Since each of the considered solutions varies in terms of area, power and performance characteristics, it would be interesting to determine how the optimal protection choices change according to design constraints. Finally, most of the previous work fails to address the overhead of recovery. It is crucial that researchers address this issue, since depending on latency assumptions and implementation, it can drastically change the relative overhead and affect the applicability of a particular protection technique.

In Chapter 3, we implement the Data-Flow Checking (DFC) protection technique for the Leon example processor core to analyze it against and in cross-layer combinations with efficient hardware and software-level techniques. Our work focuses on evaluating this solution in terms of coverage, area and performance to determine whether it can provide an optimal protection for the control-flow related structures in the core in comparison to generic, hardware techniques, such as hardening and parity, that we studied in Chapter 2. Moreover, we design an experimental setup for the Illinois Verilog Model (IVM), a more complex processor core, to obtain corresponding estimates of DFC coverage from the infrastructure provided by UT Austin. For more comprehensive protection, in collaboration with Stanford University, we analyze DFC in combinations with hardening and parity as complementary solutions for the protection of the remaining core components. Subsequently, we develop the ABFT protection for the FFT algorithm, and together with other applications

provided by Stanford University, evaluate the ABFT approach in combination with DFC. Finally, we implement a recovery solution specific to DFC and evaluate its impact on the efficiency of this protection technique. We conclude our work with a discussion of the relative protection efficiency of DFC alone and in the context of a cross-layer approach.

1.5 Cross-Layer Protection in an Accelerator

While our work in the previous two chapters focuses on resilience analysis for general-purpose cores, it is useful to extend this study to a special-purpose accelerators. Accelerators use different types of special-purpose units to speed up individual instructions (co-processors, arithmetic units), entire algorithms (ASICs) or general computation (GPUs) in the general-purpose processor. Because of the much higher computation efficiency, these architectures allow maximizing performance within a fixed power limit. While the generic resilience techniques, such as hardening, parity and replication, that we studied in the previous chapter, also apply to accelerators, only the latter two have been used. However, since accelerators perform particular functionality, they could potentially benefit more from hardware Algorithm-Based Fault Tolerance (ABFT) solutions, that we studied in Chapter 3 in software form. Similarly to general-purpose processors, accelerators can significantly vary in terms of the number of processing units or their capability for given design constraints. However, the higher component utilization and relatively smaller area make these architectures more vulnerable to errors and more sensitive to overhead for the same functionality performed. It is therefore important to evaluate the aforementioned resilience solutions to broaden the design space for accelerators and to better understand their relative benefits in the context of special-purpose cores.

The above study performed on an example accelerator would allow answering several questions related to the resilience design space. First, researchers should evaluate the vulnerability of the accelerator unit with respect to the characteristics of the underlying hardware. Special-purpose architectures generally employ a significantly smaller amount of circuitry and operations than software implementations in general-purpose processors to accomplish the same computation task. This increases the relative vulnerability of these units requiring additional protection at a proportionally higher overhead. However, another result of this could be the increased efficiency of high-level solutions, such as ABFT, in terms of protected circuitry. Subsequently, the area of the various resiliency solutions should be evaluated, since accelerators are more sensitive to this overhead due to their small area and many possible optimizations. Finally, researchers need to determine how the efficiency of hardening and ABFT compares to traditional solutions such as parity and replication. Such study could use a more sophisticated example algorithm, such as the FFT. This is a very popular algorithm that involves a large amount of shared, exchange and propagated data, which translates to increased

vulnerability and stronger detection requirements. Moreover, the simple and the more sophisticated ABFT solutions for FFT represents two extreme ends of the overhead spectrum [10].

In Chapter 4, we implement hardening, parity, replication as well as ABFT, in particular, for an accelerator core performing the Fast Fourier Transform (FFT) algorithm. The hardening and parity are implemented based on [14] and [20], respectively, while ABFT according to [10], all in the context of an FFT core obtained from DigitalFilter [21]. The core is capable of transforming a complex 1024-point integer sequence. The developed infrastructure involves modifications to the core for better integration, synthesis as well as the implementation of the resilience techniques, recovery mechanism and the fault injection framework. We first evaluate the vulnerability of the FFT accelerator with a variety of input sequences via fault injection. Subsequently, we determine the overhead of the protection features for different area-optimizing configurations in the core and the checker which involve reducing the number of processing elements and their complexity. Finally, we analyze the coverage provided by hardening, parity, replication and ABFT, with reference to particular accelerator features involved in the computation, as well as the vulnerability of the checkers. To conclude our results, we provide discussion on optimal protection choices for different configurations of the FFT core in terms of area, power and performance.

Chapter 2

Characterization of the Protection Overhead

2.1 Overview

The problem with the electrical vulnerability of transistors to radiation that leads to the transient phenomenon of soft errors has been known to researchers for a long time [22]. However, it has mainly been a concern for massively parallel products, where errors accumulate, or accuracy-sensitive [5] and safety-critical [6] systems, where the cost of failure is high. Because these are usually premium priced products for niche markets, they mainly employ the traditional coarse-grained approach of parity and replication for logic circuits in addition to error detecting/correcting codes for storage [12]. However, as feature sizes decrease and the complexity of processors increases, the individual transistors as well as the entire system become more vulnerable to faults. It is now expected that more processors and their components will need to be protected while having to address the growing overhead of resiliency features. It is therefore becoming necessary to better understand the fine-grained component-level vulnerabilities to determine how they can be addressed more efficiently at that level with a range of existing choices to optimize the overall core-level protection efficiency. Moreover, it is important to determine how the overheads of these solutions change when a stronger or more comprehensive coverage is required while possibly affecting the optimal design choices.

In Chapter 2, we address these concerns by developing Svalinn [23]. It is a modeling framework capable of illustrating area, power and delay overheads of popular resilience techniques at the level of the core or its components in the OpenRISC [24] example processor. Svalinn derives its data from a synthesis of different combinations of the protection techniques that we implement for the core. These techniques include

parity, error detecting/correcting codes (ECC) and replication, the most common solutions traditionally used in commercial processors, as well as hardening [13] [14] [15] and arithmetic codes [19], the more efficient alternatives that have also been implemented in products. The analysis that we obtain from Svalinn improves the understanding of overhead sources by illustrating how they vary depending on the proportions of different types of cells (sequential, combinational) across protected core components. Subsequently, we break down the resilience techniques into checker and data components to provide a more detailed understanding of unit costs associated with each solution. Moreover, we also use Svalinn to illustrate how the most optimal coverage is achieved at the level of the core or its components by applying protection techniques according to their best fit.

Despite extensive prior work on efficient hardware resilience, we are not aware of any prior work that compares the characteristics and trade-offs of different protection techniques across core components on a single platform. We use the OpenRISC platform as a case study, since it is representative of simple cores that are becoming popular in throughput-oriented, power-aware and embedded designs [25] that are also more sensitive to protection overhead. The first goal of our work is to improve the general understanding of the resilient design with the existing solutions via a detailed analysis of our example processor. This should help researchers reason about these concepts in other architectures that may vary in configuration. The second goal is to illustrate the limits of the currently used resiliency techniques to motivate exploration of the more efficient alternatives that leads to our work in Chapter 3. While Svalinn is capable of modeling any configuration and range of protection, for the ease of explanation, we present comprehensive designs in this chapter. The evaluation of the error coverage and recovery is done in Chapter 3 only for the select techniques that we explore there further. This chapter makes the following contributions.

1. We develop a set of popular protection techniques for a single platform, an example OpenRISC processor, to provide a baseline for a fair comparison.

2. We apply the above resiliency solutions to individual components in the core to better address vulnerabilities and understand overheads at that level.

3. We analyze protection techniques in terms of their storage logic components to better understand sources of these overheads that hold for all architectures.

4. We illustrate how the overall core-level protection can be optimized by applying a variety of solutions according to their best fit and design constraints.

5. We use this analysis discuss the efficiency of the traditional resilience solutions and their more efficient alternatives, which leads to our work in Chapter 3.

This work was published in the IEEE Micro article in 2013 [23].

2.2 The Soft Error Problem

In this section we give the overview of the soft error problem. The trend of continuing technology scaling makes circuits more susceptible to soft errors, transient phenomena that are caused by upsets due to charged particle strikes [1] (Fig. 2.1) or by timing violations due to process variability [2]. Specifically, smaller feature capacitance makes it easier for an induced charge to change the logic state at a given, now decreased, operating voltage. The decreased feature size also exaggerates lithographic imperfections that result in varied timing characteristics across the circuit. Moreover, the radius of a single particle strike now covers a larger part of the circuit, potentially giving rise to multi-bit errors [8] (Fig. 2.2). The growing power consumption in the core necessitates employment of drastic power-saving techniques such as dynamic voltage and frequency scaling (DVFS) to fit within the fixed power limit. These techniques exacerbate the soft-error problem by lowering critical charge, prolonging exposure and potentially causing timing violations [11]. Furthermore, the increasing complexity of workloads makes entire systems more vulnerable due to error accumulation and increased probability of affecting the output. Today, the primary source of charged particles are cosmic rays, as opposed to alpha particles from chip's packaging. The raw particle rate at the sea level is about 1e-10 per hour per cm$\hat{2}$ and it increases over 2 times for every 1000 meter increase in altitude due to the decreasing protection from Earth's atmosphere. Therefore faults become even a more significant concern at airplane cruise altitudes and in space where the radiation is more than 2 orders of magnitude higher than on earth [1]. In a design setting, soft errors are emulated through error injection or exposure to a particle beam [26].

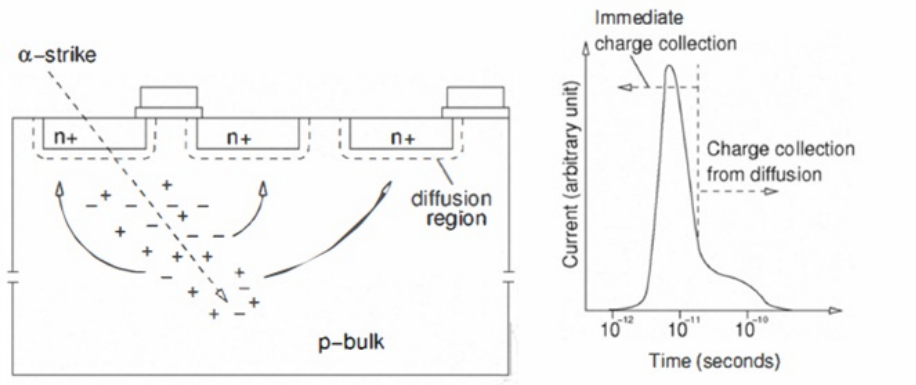

Figure 2.1: Alpha particle strike in an NMOS transistor and the resulting current pulse [1].

The recent advances in fabrication technology, such as multi-gate transistors [3] and silicon-on-insulator (SOI) [4] are expected to decrease the soft error vulnerability by an order of magnitude for the next transistor

generations. However, the trend of increasing error rates inevitably continues with further scaling. Soft errors are a concern mainly for sequential logic (flip-flops) and SRAM arrays (such as those in the register file, cache, and memories) [3]. These circuits include cells that implement a feedback loop that maintains their state which can be then easily flipped and reinforced in the case of a particle strike. However, most faults that occur in the system are naturally hidden by some form of masking that is inherent to the circuit or the function it performs. If particular data is not used or does not affect the correctness of the result, the error is masked architecturally or logically. In the case of combinational logic, made of gates, the charge that is constantly supplied from the power supply can overwrite most of the transient charge upsets, an effect called electrical masking. Moreover, if a fault in combinational logic does not appear in a flip flip at the clock edge, it is temporally masked. Although much less significant, errors in combinational logic also increase with the decreasing feature sizes [3]. Depending on the circuit size and fabrication process, various sources report the measured resulting soft error rate on the order of 1-2 per year for a single processor core [1] and 3-5 for a single SRAM chip [27] at the ground level. It is therefore easy to project that, without soft-error protection, these numbers significantly increase for larger multi-chip systems [28] or at higher altitudes.

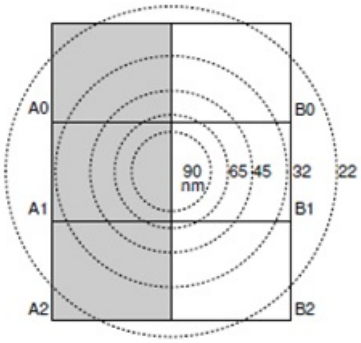

Figure 2.2: Area covered by a 2um particle strike radius with respect to the area of two 3-bit registers at various technology nodes [8].

2.3 Previous Work

In this section we briefly summarize previous work on soft error protection techniques and recovery methods. There is a variety of existing and proposed techniques that span architecture, software, and hardware level. At the cell level, various types of hardening [13] [14] [15] redesign structures of flip flips and SRAM cells to help maintain the correct logic level in the case of an upset. Razor technique employs a delayed shadow latch that corrects the main latch in the case of a timing violation [29]. At the logic level, parity protects a bit word by verifying its binary properties. Error detecting/correcting codes (ECC) can restore multiple bits

in a word with a use of a checksum [12]. Arithmetic codes (parity prediction and residue) detect errors by comparing properties of inputs to those in an output [19].

At the architecture level, spatial redundancy replicates core or its units for redundant execution [12]. Monitor cores repeat or verify a subset of the main core's functionality [30]. Data and control-flow checking (CFC/DFC) [16] [17] [31], including their software implementations [32], verity correct transitions between instructions and basic blocks in the program. At the software level, various forms of temporal redundancy repeat the execution of individual instructions, their groups or entire workloads [33] [34]. Redundant Multi-Threading (RMT) replicates a thread of execution in a superscalar core while combining the concepts of spatial and temporal redundancy [35]. Symptom-based solutions detect abnormal behavior in an application to detect hardware faults [36]. Algorithm-based fault tolerance (ABFT) takes advantage of algorithm properties to verify computation [18] [10].

There are several forms of recovery with various applicability to the above protection techniques. At a coarse-grained level, the workload can be squashed and restarted with a loss of performed computation. Software check-pointing can be used to store application's state that is restored with a roll-back on error detection with a partial loss of performed computation. Various buffering allows storing of the results to postpone the result commit until the check is performed according to detection latency of the underlying resiliency solution. At a fine-grained level, core's recovery mechanism can be used to flush the pipeline if error is detected within the first few of its stages [5].

Since the soft error problem has been mainly relevant for the high-end commercial products in the past, the industry has not moved beyond using traditional approaches such as coarse-grained replication and parity for logic circuits, ECC for storage as well as hardening and arithmetic codes in a few cases. Moreover, there is no previous work that would evaluate trade-offs between these popular solutions when applied to different components in the same baseline processor. Therefore there is little understanding of their relative overhead and how effective they are at addressing higher error rates.

2.4 Analyzed Protection Techniques

In this section, we give more details about protection techniques that we implement and evaluate in this work, including their functionality and applicability to different types of circuits and core units. Parity, replication and ECC are the most common resiliency solutions traditionally used processors, while hardening and arithmetic codes are the more efficient alternatives that have also been commercially implemented.

Hardening: This technique provides soft-error protection for storage circuits (SRAM) and sequential logic (flip-flips). The approach involves adding redundant transistors to storage cells to provide redundancy

that reinforces the logic state [13] [14]. Other variations implement a feedback path that cancels the effect of an upset [15]. Stronger hardening designs might be required in case of more intense radiation or single-event-multiple-upsets (SEMUs). This solution is implemented at the library cell level with no changes to architecture or logic in the design. Our modeling of this technique is based on [14].

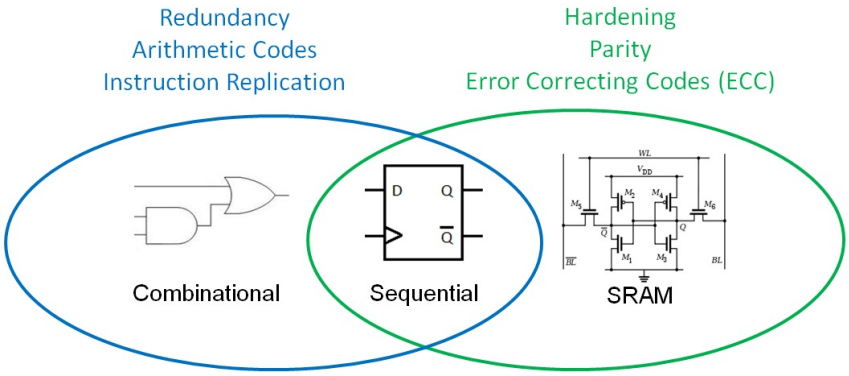

Figure 2.3: Applicability of protection techniques traditionally used in products and analyzed in our work.

Parity and ECC: Both parity and ECC can protect storage circuits (SRAM) and sequential logic (flip-flops) against soft errors [12]. Parity provides single-bit error detection, whereas single-error correct, dual-error detect (SECDED) provides detection of double-bit errors or correction of a single bit (Fig. 2.4). Although not considered here, alternative variations include interleaved or 2D parity/ECC [37] and, rarely used, dual-error correct, triple-error detect (DECTED) for improved protection at a higher overhead.

Arithmetic codes (parity prediction and residue): Parity prediction detects single-bit soft errors in the results of some arithmetic operations, usually in the integer arithmetic logic unit (ALU), by comparing parity of the operands against that of the result. Residue codes, on the other hand, detect errors that translate to any number of bit flips in the result, for a larger set of arithmetic operations, usually in the floating-point unit (FPU), by comparing modulo residue of the operands to that of the result [19] (Fig. 2.4). We implement and apply these selectively according to [19] and leave further exploration for future work.

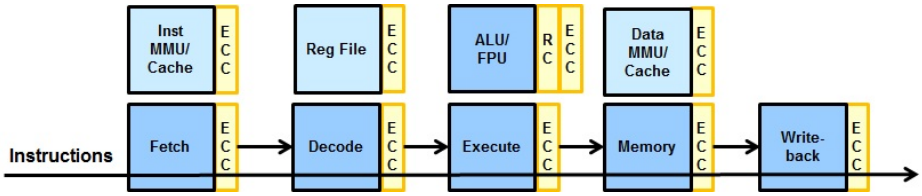

Figure 2.4: Protection of pipeline components with codes. Memory arrays and sequential logic can be protected by parity or ECC, while arithmetic units optionally via arithmetic codes.

Replication: This technique protects sequential and combinational circuits against soft and hard errors. Each instance of the unit, at the level of the core or its components, executes the same operation in parallel

in lock-step (Fig. 2.5 while comparing results every cycle [12]. This technique can recover from any number of transient faults within the replicated component, as long as both copies are not affected (unlikely due to physical separation). Although not considered here, variations include relaxed lock-step operation and and checking at multi-cycle time intervals (possibly overlapped with computation).

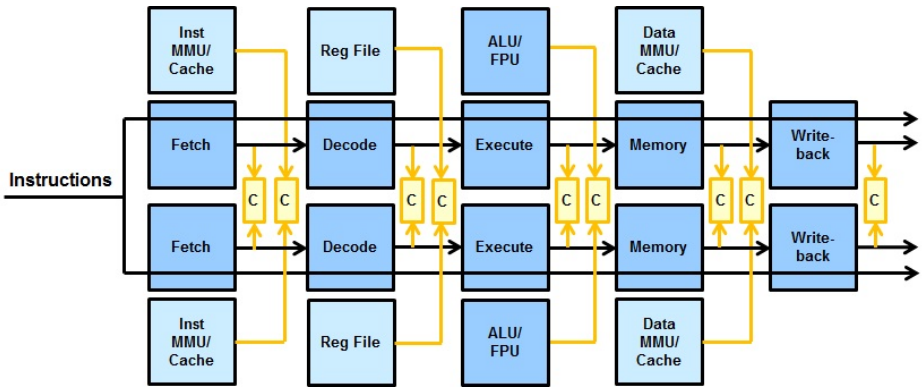

Figure 2.5: Protection of pipeline components with replication. The implementation includes a redundant unit and a comparing checker.

Instruction replication (temporal redundancy): This core-level technique sequential and combinational circuits against soft errors. In our model, we assume that either hardware or software-replicated instructions are executed back to back in the pipeline [33] (Fig. 2.6). The results are saved into an ECC-protected buffer in a compressed form [38] and compared every cycle. This technique protects against any transient faults within the duration of repeated execution. Although not considered here, variants include Redundant Multi-Threading (RMT) [35] in a superscalar core and checking at multi-cycle time intervals.

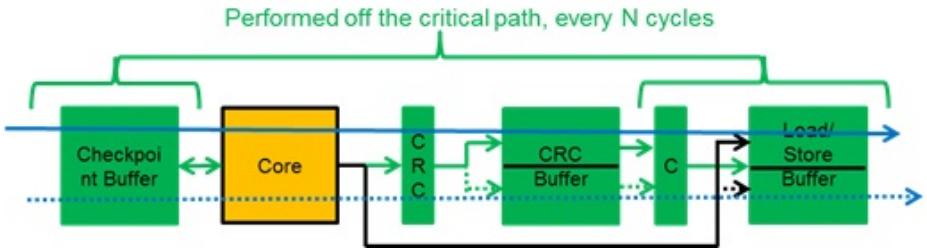

Figure 2.6: Protection of execution with instruction replication. Checksum and checkpoint units are required for checking and restoring, respectively

2.5 The OpenRISC Platform

In this section we describe the architecture circuit characteristics of the OpenRISC platform. This processor is a simple 32-bit architecture that features a scalar, in-order pipeline, static branch predictor [24]. The

synthesis results show that the caches, the execute (includes the ALU) and the fetch units are the largest contributors to the core area 35%, 24% and 15%, respectively (Fig. 2.7a). The register file has a larger implementation to minimize delay because there is no dedicated register access stage. Since flip-flops and SRAM arrays are vulnerable and require protection, their relative contribution in the circuit translates to the overall protection overhead. Logic circuits (flip-flops and gates) contribute more than half (57.8%) of the overall core area, of which 34.5% are flip-flops (Fig. 2.7b). The execute and decode units have the largest number of flip flips of all components (29%, 24% of all cells, respectively). Note that flip-flops are larger than gates. The relative distribution of area, delay, and power across core components is important when considering trade-offs between protection solutions and the overall core-level overheads. The execute stage (ALU) has the longest delay which sets the critical path for the core. While caches and the register file have the largest power per access, larger parts of units such as decode and ALU are utilized leading to increased power consumption (Fig. 2.8).

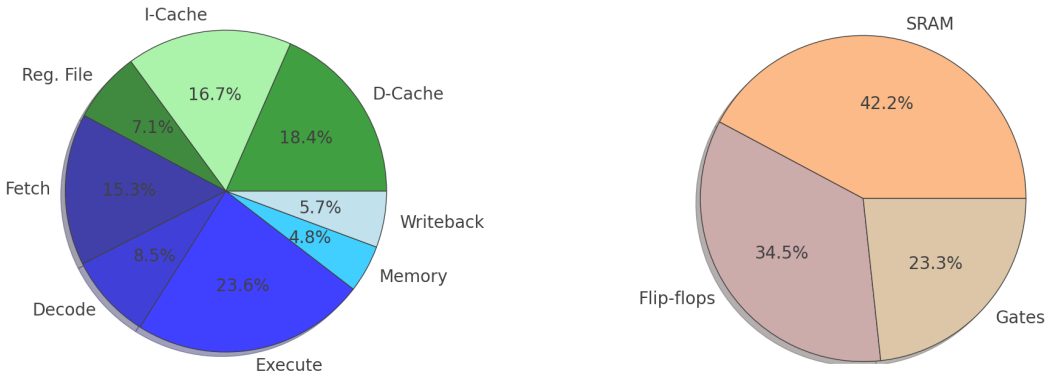

Figure 2.7: Area composition of the OpenRISC core according to: a) components and b) types of cells. Most of the area consists of vulnerable SRAM and flip-flop cells.

2.6 Methodology

In this section, we describe the methodology for our work. For the purpose presenting the results with a focus on pipeline stages, we use an embedded-system-like configuration for the OpenRISC with 1kB caches as well as disabled MMU (not needed for single workload) and FPU (using software instructions instead). The considered protection techniques including parity, error detecting/correcting codes (ECC), replication and arithmetic codes (parity prediction are residue) are implemented in RTL and integrated with the OpenRISC core. For hardening, the overheads per cell are obtained from previous work [13] [14] [15] since the cell library was not available. Instruction replication is estimated by roughly doubling the number of executed instructions [33]. These resiliency solutions are synthesized individually and in combinations to obtain area,

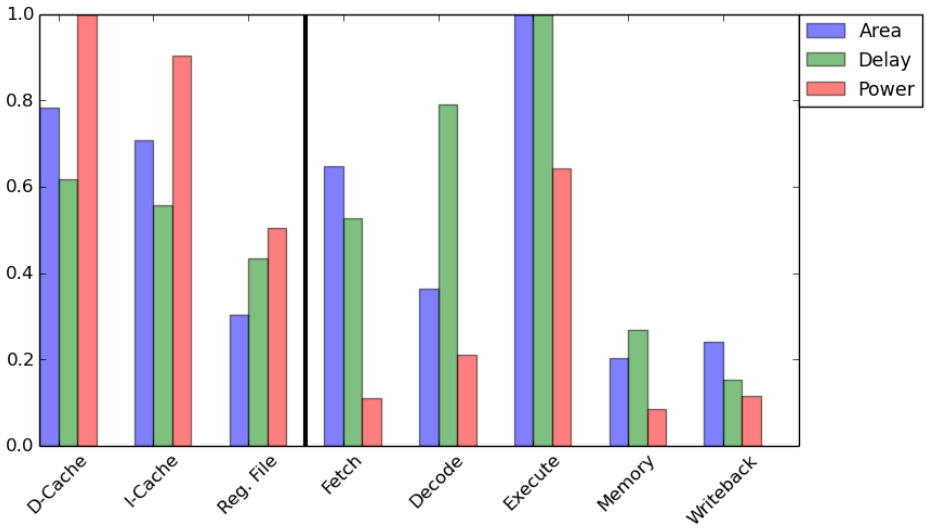

Figure 2.8: Relative area, delay, and power per core component (normalized to the largest for each metric). Caches and execute stage (ALU) are the largest in area. The ALU and the decode stage have the longest delay. More utilized components use more power.

power and delay data which are used as comparison metrics. The synthesis is performed with a 90-nm library using Cadence tools [39]. Our results are included in the Svalinn [1] framework capable of displaying them for many core configurations as well as making projections for other architectures. We augment Svalinn analysis with energy data obtained by characterizing average activity over regions of interest in a set of SPEC2000 benchmarks [40] using gem5 simulator [41]. Cacti framework is used to model buffers and SRAM arrays [42]. The evaluation of coverage and recovery is left for detailed exploration in Chapter 3 for a select set of solutions analyzed there.

2.7 Component-level Analysis

In this section we show the first set of important result obtained from our Saving framework. It includes the analysis of per-component area, delay and energy characteristics (Fig. 2.9) as well as the corresponding protection overheads in the Operatic core (Fig. 2.10). This data helps visualize where each resiliency technique is a good fit depending on component size, functionality or cell types. Although Svalinn can model any range of protection, for the ease of explanation, we present the upper-bound overheads that are representative of comprehensive protection. Moreover, we express numbers as sums or averages (delay) measured across all components where techniques are applicable while including comments regarding relative differences.

[1] In Norse mythology, Svalinn is a legendary shield that stands before the sun.

2.7.1 Area

Depending on the type of hardening, the area of the protected flip-flop is roughly doubled, and increased even slightly more than that for the SRAM cell. Because almost every such cell must be hardened for complete coverage, this technique clearly incurs a high area overhead for storage components (91%) compared to that for logic components (54%) (Fig. 2.9) when applied where applicable across the core. Moreover, since some flip-flops in logic circuits use larger transistors for improved drive current, they incur even higher overhead.

Codes (parity and ECC) are very efficient at protecting storage circuits (SRAM), where their cost is amortized over long words (32 in this case), check on infrequent access and shared encoders/decoders per read/write port. Their area overheads is 10% and 38%, respectively (Fig. 2.9). The opposite holds for logic circuits, where these techniques incur 42% and 142% overheads due to narrow words (8 to 32 bits) and each evaluated at every cycle with its own encoder/decoder. The area contribution of encoders/decoders always dominates the overhead of protected logic circuit, especially for small components and simple cores such as OpenRISC (Fig. 2.10). This makes ECC techniques prohibitive while favoring parity there. Although it sacrifices coverage capability, parity incurs lower overhead than hardening due to better amortization and low checker size.

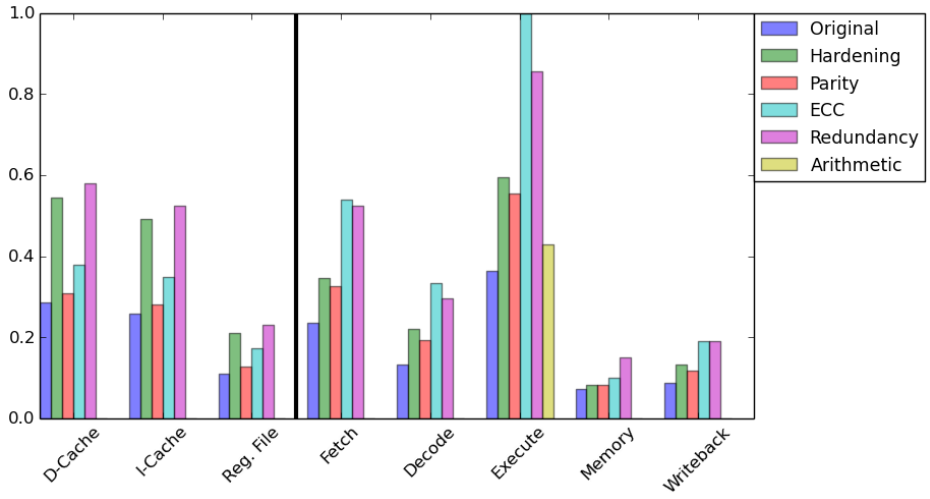

Figure 2.9: Relative area per protected core component (normalized to the largest). Parity and ECC incur significantly more overhead for logic components, in the pipeline and arithmetic units, than SRAM arrays.

Although replication is the most expensive protection technique in terms of area, due to the redundant copy of the protected unit, it expands its coverage while including combinational logic as well. It is an important gain when compared to ECC for logic circuits which has only a slightly smaller overhead in a comprehensive configuration (123% and 142%, respectively) (Fig. 2.9). Moreover, the solution can decrease the size of the checker due to a smaller number of signals compared at the boundary of the unit in the case of

logic circuits. This can be contrasted with codes, where every flip-flop is verified by the decoder regardless of its contribution to the output. Although impractical, replication could be applied to storage circuits with an overhead of 105% across the core due to a smaller number of words checked on access, compared to logic circuits where the overhead is 123%.

Arithmetic codes (parity prediction and residue) remain the most area-effective for select operations in the ALU and the FPU (15% and 18%, respectively) (Fig. 2.9). The checking of the output alone has a potential of covering a large number of flip-flops involved in its computation. These techniques have to be complemented by other techniques to cover the remaining functionality in arithmetic units.

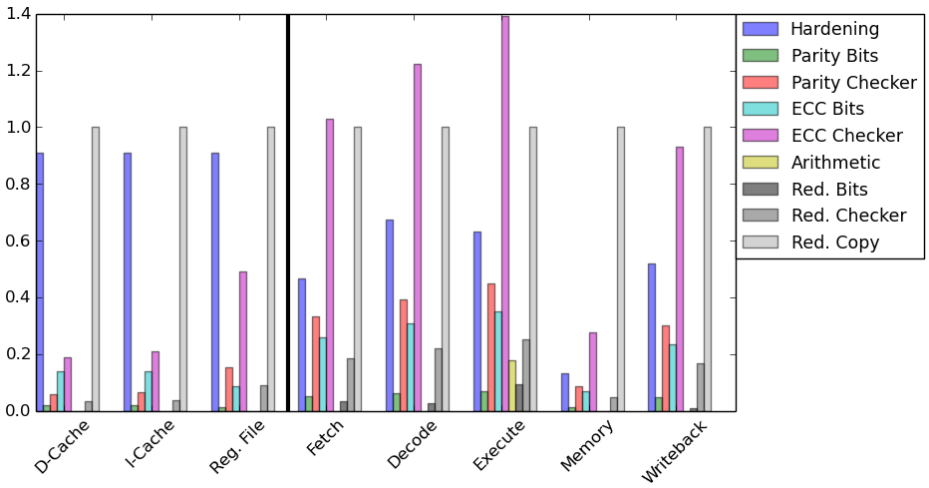

Figure 2.10: Detailed area overhead of select protection techniques per core component (normalized to unprotected). Encoders/decoders and checkers have a significant impact on the area, especially for the small OpenRISC core.

2.7.2 Delay

Svalinn shows that cell-level changes introduced by hardening translate to a negligible 0.1% delay increment, on average, for both logic and storage circuits, because there is usually only a single hardened flip flip at the end of each each components critical path (Fig. 2.11). ECC, parity, replication and arithmetic codes, on the other hand, contribute significant delays of 22%, 16%, 14% and 0% (not applicable), respectively, for storage and those of 13%, 9%, 7%, 6% respectively, for logic circuits. This is due to additional circuitry, such as encoders, decoders, residue/prediction generators, and comparators, in the critical path. This delay contribution is larger for storage units (SRAM) with larger bit words to compare as well as where ECC is used, due to a more complex checker. However, in most cases these delays are hidden by overlapping with the delay of the slowest component in the system, unless protection is applied there (Fig. 2.8). Some designs pipeline parity and ECC to decrease the critical path delay while increasing detection latency [20]. The delay

overheads are relatively more pronounced in components with short critical paths or simple cores such as OpenRISC. Since the execution unit is by far the slowest component, the application of protection elsewhere would not change core's critical path.

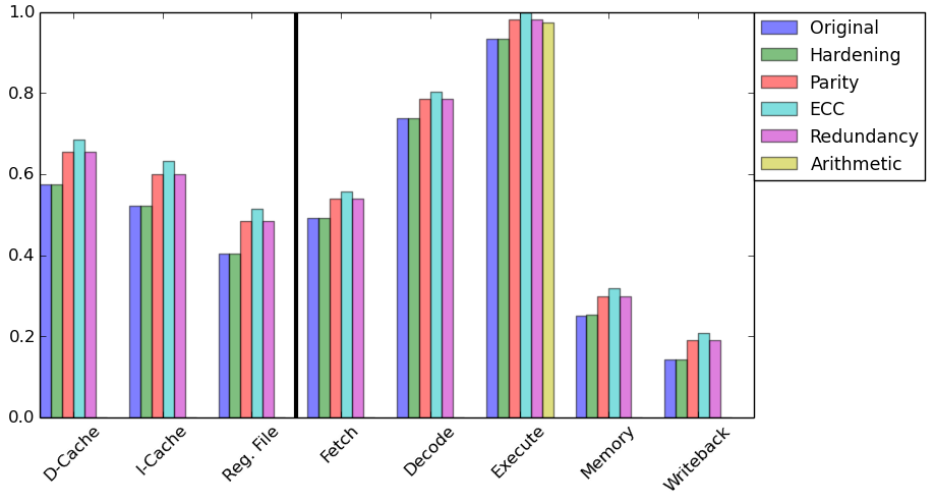

Figure 2.11: Relative delay per protected core component (normalized to the largest). Protection overhead of the ALU affects the core frequency. Elements with shorter critical paths are disproportionately affected, especially when using ECC.

2.7.3 Power and Energy

Svalinn shows that the power consumption that protection techniques contribute to core components is proportionally similar to the area they contribute (Fig. 2.12). That is because all of the protection circuitry is utilized on a check. As such, the added protection does not significantly change the relative power consumption between components. Hardening, parity, ECC, arithmetic codes and replication add 93%, 11%, 39%, 0% (not applicable) and 108% consumption to storage circuits, respectively and 58%, 45%, 150%, 20% and 125 % consumption to logic circuits, respectively. The high activity of pipeline stages during benchmark execution, as we measured in the gem5 simulator, increases their energy consumption with respect to storage components, especially in the case of ECC and replication, solutions with higher power dissipation.

2.8 Core-level Analysis

In this section we present another important analysis obtained from Svalinn framework. It illustrates the trade-offs between comprehensive core-level protection choices for the OpenRISC core. Our results illustrate how much overhead can be saved by applying combinations of techniques that are the best fit for particular core components (Fig. 2.13). For the ease of explanation, we present results in terms of area while discuss

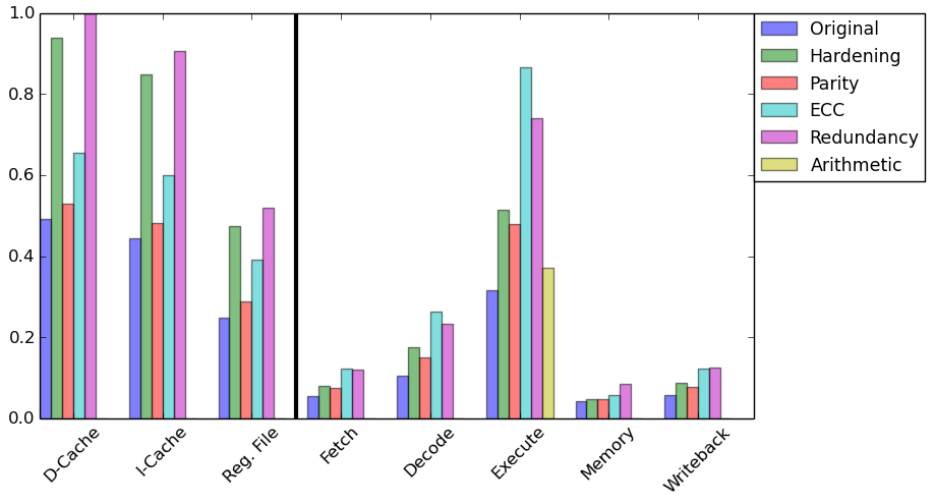

Figure 2.12: Relative average power per core component (normalized to the largest). More active components such as the instruction cache and some of the pipeline stages incur the highest energy easily exacerbated by expensive protection.

delay and energy metrics. We summarize characteristics of these techniques as sums or averages (delay) across all applicable and comprehensively protected components with comments regarding particular circuit types and core units.

2.8.1 Individual Protection Techniques

To determine the relative overhead of protecting the core with each individual resiliency technique, we used Svalinn to evaluate several scenarios that apply each technique to all relevant components in the core (Scenarios 2 through 7 in Fig. 2.13). Although some are commonly used, these establish a baseline for evaluating combined techniques in the next section.

Scenario 1 corresponds to the original design with no soft-error protection implemented. Individual techniques in Scenarios 2 through 4 protect only storage and sequential logic circuits. Because of the large contribution of storage circuits, hardening increases the overall core area more than parity (by 69% and 28%, respectively). However, hardening contributes a shorter delay to the core than parity (0.1% and 5%, respectively), while larger increase in the core power dissipation (80% and 21%, respectively). When protected entirely by ECC, the core suffers from high area, delay, and power overhead (98%, 7%, and 75%), mostly due to protection of the state elements in logic circuits, where ECC is suboptimal in area and coverage. ECC has the largest increase to the cores cycle length (in the ALU, the slowest component) of 7%, as well as in energy consumption due to the high pipeline stage activity. If a lower resiliency is acceptable, parity clearly proves to be a more practical alternative for some of the smaller logic components, due to a small

checker size. In Scenario 5, only select operations in arithmetic units (ALU and FPU) are protected by parity prediction and residue codes, respectively. The core-level area, delay, and power overheads for this solution are 4%, 5%, and 3%, respectively. In contrast to previous scenarios, Scenario 6 represents protection of all storage and logic circuits with replication. This scenario suffers from high area and power overheads (116% and 112%, respectively), largely due to replicated storage circuits where this solution is suboptimal in area and power. Replication imposes a smaller delay of about 5% to the cores critical path. Scenario 7 provides coverage via repeated execution of individual instructions. Although this scenario offers protection at moderate additional area and power (19% and 18%), it incurs a significant performance and energy cost due to temporal redundancy. This loss can be partially reclaimed when using multi-threaded super-scalar cores [35]. The variant of this solution presented here checks correctness at the granularity of 100 cycles by compressing memory stores into a checksum. It includes circuitry and a buffer for checksum generation, as well as the buffer to save memory store values before commit on a check. The coverage in each scenario varies depending the amount of circuitry covered by the technique and its detection capability.

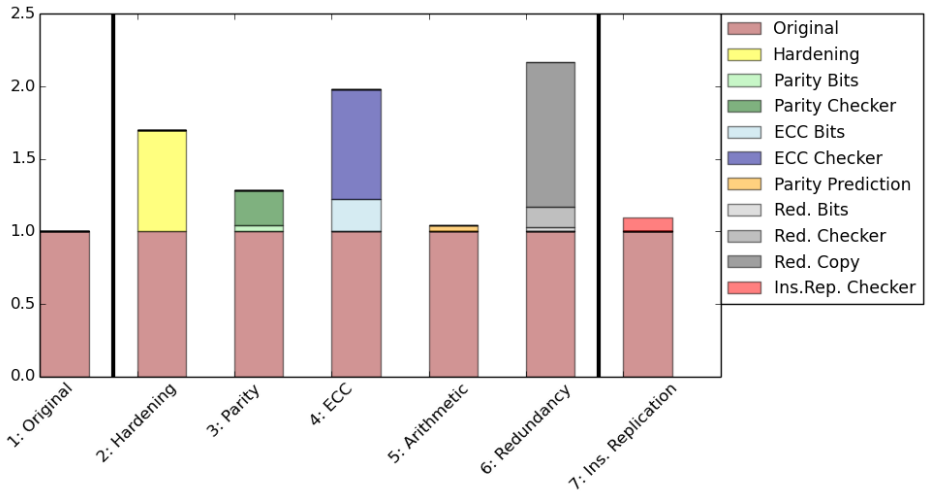

Figure 2.13: Relative core area per scenario where individual techniques are used (normalized to unprotected). Replication and ECC (for logic circuits only) incur the highest overheads in area while instruction replication in performance.

2.8.2 Combined Protection Techniques

The results obtained from Svalinn clearly illustrate how the use of a single resiliency technique results in unnecessary excessive overhead. Therefore, we use Svalinn to evaluate further protection scenarios that apply combinations of different resiliency mechanisms to relevant core components according to their best fit (Scenarios 2 through 9 in Fig. 2.14).

Scenario 1 corresponds to the original design with no soft-error protection implemented. For the remaining scenarios, we assume that ECC is the most optimal solution for SRAM components where it has been traditionally used, due to its moderate overhead there and acceptable coverage. Therefore in these scenarios, we apply various combinations of the remaining techniques only to logic components, to observe the varying impact on the overall area of the core. In Scenario 2, replication is used for all logic components to provide comprehensive coverage there at the 88%, 5% and 68% overall core-level area, delay and power overheads, respectively. Replacing replication protection with less capable but more efficient parity prediction for the ALU, in Scenario 2, reduces these footprints to 61%, 5% and 47%, respectively. Applying hardening comprehensively to logic components, in Scenario 3, protects only flip-flops there while decreasing the protection overhead to 47%, 0.1% and 45%, respectively. Replacing this protection with parity prediction for the ALU, in Scenario 4, furthers lowers the area and power footprint to 36% and 37% while increasing the delay to 4% (since ALU is the slowest component). Although even less capable than hardening, applying parity for the protection of flip-flops, in Scenario 5, reduces the protection area, delay and power footprint to 36%, 4% and 37%, respectively. Similarly, application of parity prediction to the ALU, in Scenario 7, further decreases the area and power overhead to 32% and 34% while increasing the delay to 5%. If performance can be traded for area, instruction replication can be used for the comprehensive protection of the execution, in Scenario 8, flow while lowering the checker footprint to 25%, 4% and 33%. The coverage of this solution for the ALU can be slightly improved by adding parity prediction there, in Scenario 9, at the moderate increase in overheads to 30%, 4% and 36%. However, even in terms of area, scenarios 8-9 are less competitive compared to scenarios 6-7, for example, due to large implementation of the checker in a simple core such as OpenRISC. As mentioned above, the coverage of each scenario varies depending the properties of individual resilience technique and corresponding amount of protected circuitry. However, in terms of their general protection capability, they are arranged from the strongest (Scenario 2) to the weakest (Scenario 7), which corresponds to the decreasing overhead across these solutions.

2.9 Discussion

The increasing errors rate in sequential logic is expected to necessitate more comprehensive coverage in the core, including logic circuits that often lacked protection in the past. While still much less significant, errors in combinational logic are expected to increase as well while contributing to the problem. Researchers have not yet explored the relative overheads of different techniques, and various combinations, in this context. Svalinn shows that, beyond parity and ECC, in Scenario 6 in Fig. 2.14, achieving more comprehensive coverage with replication might increase the cores area by 34% with hardening and replication in Scenario 2. The increase

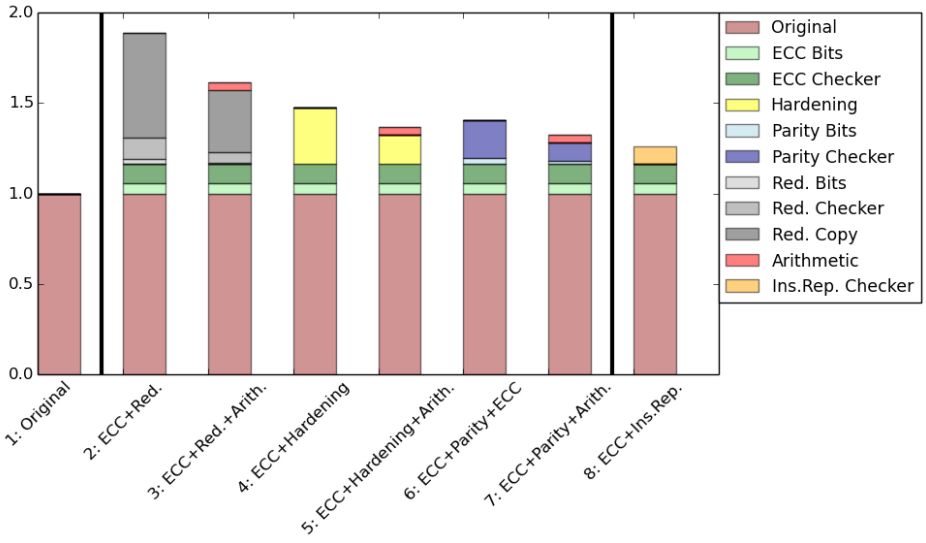

Figure 2.14: Relative core area per protection scenario where combined solutions are used (normalized to unprotected). Area and power are minimized using hardening, parity and arithmetic codes for logic while ECC is preferred for SRAM arrays.

in energy is expected to be similar, while the critical path delay might grow if a form of parity prediction is additionally used for the ALU, the slowest component. More complete coverage might need to protect the checker circuitry (to avoid false negative detections) at an additional cost.

In addition to more comprehensive protection, the increased error rates and the possibility of multi-bit errors necessitate stronger coverage. Among existing solutions, both hardening and replication are capable of protecting against such errors while providing higher overall detection since these solutions strengthen or replicate each of the protected cells or units, respectively. While parity, ECC and arithmetic codes on the other hand, can protect against single-bit errors, they generally have lower coverage since they collectively protect groups of cells with a single checksum. However, interleaved ECC is a stronger and acceptable solution for SRAM arrays. Other, more specialized, architecture and software solutions such as Data Flow Checking (DFC) and Algorithm-Based Fault Tolerance (ABFT), considered in Chapter 3, protect aspects of the program flow which naturally covers relevant ranges of single and multi-bit errors in sequential and combinational logic. In terms of the resiliency improvement, upgrading from a solution based on parity and ECC (Scenario 5 in Fig. 2.14 to those with hardening and parity (Scenarios 4 and 2) might increase the protection overhead from 5% to as much as 34%, respectively. This increment is mostly due to the high overheads of components increased in size or replicated by the two solutions, respectively.

The results obtained from Svalinn help us reason about the efficiency of the currently used protection when addressing potentially higher error rates. We expect that storage (SRAM) components to continue benefiting from comprehensive protection via ECC that can easily amortize the overhead and provide multi-bit coverage.

However, replication may no longer be cost effective for the protection of logic circuits at doubled overhead as shown in Fig. 2.14. While parity is still a viable, but less capable, solution there, we expect to see more interest in exploring more efficient existing alternatives, such as selective hardening and arithmetic codes, that are not yet in widespread use. This is further motivated by fact that popular simple cores such as OpenRISC experience more pronounced overheads due to the small size and large flip-flop cell contribution. In terms of the design approach, researchers need to find more accurate ways of determining vulnerability so that it can be addressed selectively while maximizing coverage. Beyond the existing protection techniques, it is necessary evaluate other, proposed, competitive protection alternatives such as architecture-level data flow checking or software-level algorithm based solutions.

2.10 Conclusions and Future Work

In this chapter we performed a detailed analysis of the area, delay and power overheads of the commonly used protection techniques. The data from our analysis is included in Svalinn, a modeling framework capable of presenting various protection configurations that we study. This work serves as an aid to improve the general understanding of trade-offs between existing protection choices by comparing them on a single processor platform. Our work helps understand how these overheads change, depending on types of protected circuits and units, when more comprehensive solutions are used. For this purpose, we study characteristics of protection techniques per core component, break some of the techniques further into data and checkers as well as analyze core-level protection scenarios consisting of individual or combined solutions.

Another goal of our work is to illustrate the challenges and limits of the existing protection techniques and the approach to the resilient design, especially when targeting higher error rates. We achieve this by presenting relative overhead increase between the commonly used solution (based on ECC and parity) and those that provide more comprehensive coverage for logic circuits (hardening and replication). While we expect ECC to remain an optimal protection for storage (SRAM) units, the more complete protection of logic circuits with replication or even comprehensive hardening may not be cost effective due to the high overhead. Therefore future research directions should include finer-grained analysis of vulnerability as well as consideration of other competitive protection techniques from architecture and software levels.

Chapter 3

Resilience Optimization with Cross Layer Techniques

3.1 Overview

The current trends of device scaling, increasing design complexity, and the growing range of operating environments necessitate the use of more efficient resiliency solutions even to maintain the same (let alone stronger) levels of resiliency while minimizing the corresponding overhead. Our main focus is on soft-errors in sequential logic, which by far dominate the overall error rate. The work in the previous chapter illustrates challenges with the current approach due to the high overhead of hardware solutions, especially when those are applied according to vulnerability determined at the coarse-grained or component-level. It is therefore necessary to explore ways to combine solutions to achieve more efficient alternatives. These could involve architecture-level data-flow checking (DFC) [16] [17] and software-level algorithm-based fault tolerance (ABFT) [18] [10] which, similarly to arithmetic codes studied in Chapter 2, can protect particular core or application functionality at a potentially lower overhead. Moreover, it is also important to use more accurate ways of determining vulnerability, at the flip-flop level, for the more efficient application of the current hardware techniques such as hardening [13] [14] [15] and parity [20]. These efficient solutions, also studied in Chapter 2, could be used for the remaining parts of the core selectively for particular soft error rate (SER). Ultimately, the cross-layer combination of DFC and ABFT with the hardware-level solutions could potentially provide additional benefits. Finally, the cost of efficient local recovery needs to be incorporated in the design.

In Chapter 3, we address the above concerns by implementing architecture-level DFC technique for Leon and IVM example cores to evaluate it against and in cross-layer complementary combinations with

hardware-level hardening and parity as well as software-level ABFT. DFC is a specialized protection solution that can compress a significant amount of data flow state in a signature to be checked at run time. We determine it to be potentially competitive against generic techniques such as hardening and parity. The Leon implementation involves the full tool chain with the compiler support, binary translation, hardware checker and analysis scripts. The IVM implementation involves an equivalent simulation setup designed and executed in collaboration with UT Austin. The DFC is first evaluated against selective hardening and parity that are applied with fine-grained vulnerability analysis via error injection on the FPGA platform, in collaboration with Stanford University. Subsequently, the DFC is combined with the latter two solutions for the protection of the remaining circuitry to determine its benefit in these cross-layer combinations. Also, we develop software-level ABFT technique for the FFT application and, together with other ABFT applications provided by Stanford University, evaluate it in combinations with DFC, hardening and parity. Finally, we develop local recovery solutions for DFC in the two example cores to determine its impact on the effectiveness of this solution. Although DFC and ABFT also cover combinational logic, we are mostly concerned with sequential logic where most errors take place.

Although DFC has been previously proposed and studied, it has not been evaluated against other efficient techniques on the same platforms. We use Leon and IVM platforms because they represent simple in-order and more complex out-of-order cores. These platforms allow us to observe the impact of studied solutions on the resiliency of both architecture types via synthesis and injection, to get their flip-flop-level vulnerability. The first goal of our work is to evaluate the efficiency of DFC in relation to hardening and parity. This helps determine whether DFC, in its own protection scope, is a valid alternative for these efficient generic hardware solutions. The second goal of our work is the evaluation of the cross-layer design space that involves complementary combinations of the aforementioned techniques and the ABFT. This helps understand whether specific solutions such as DFC and ABFT, within their protection scope, can provide the resiliency benefit on top of hardening and parity. Our results show that, although DFC provides efficient resilience compared to traditional replication or other coarse-grained solutions, it is less competitive than selective hardening and parity. The latter two can target individual vulnerable flip-flops and incur small or no cost of recovery. The ABFT remains an alternative or complementary design option for a particular algorithm when performance is not constrained. Finally, we discuss the varying benefits of the considered efficient solutions and their optimal choices dependent on design constrains. This chapter makes the following contributions.

1. We implement DFC technique for Leon, with the full toolchain, and for IVM, in an equivalent form in simulation in collaboration with UT Austin.

2. We develop software ABFT solution for the FFT benchmark, as a part of a larger set of ABFT codes in collaboration with Stanford University and IBM.

3. We evaluate hardening, parity, DFC and ABFT on Leon and IVM, the same two hardware platforms, to provide a baseline for a fair comparison.

4. While less efficient than hardening and parity, we show that limited coverage of DFC would require alternative protection for most resilience targets.

5. We evaluate potential benefits of the cross-layer combinations of techniques in the context of design constraints and target soft error rate.

Parts of this work have been published in [43]. The remaining results are being prepared for the MICRO conference submission.

3.2 Previous Work

In this section we briefly summarize previous work on the more efficient soft error protection techniques and recovery methods that are relevant to our work in this chapter.

Control and data-flow checking (CFC/DFC) are architecture-level techniques which extend from the early concept of a watchdog processor that has been used to monitor various aspects of run-time behavior. CFC/DFC are specifically used for the verification of the correct instruction flow in the program. The CFC/DFC approach has been studied in academia in earlier [16] and more recent [17] [31] publications, among others. Due to numerous design trade-offs such as monitored signals, placement of the checker and signature algorithm, the scope of protection varies across these implementations. The previous evaluation of the DFC alone, in the aforementioned work, has not been sufficient to illustrate its benefits over alternative solutions on comparable platforms [17]. CFC/DFC has failed to receive much attention from the industry for the possible reasons of compiler-hardware implementation challenges, varying design trade-offs and coverage across architectures as well as possibly a non-trivial cost of recovery. Various implementations of software-level CFC, such as Control-Flow Checking by Software Signatures (CFCSS) [32], have also been proposed in the past. However, due to a significant cost of signature processing at the software level, these can only protect branch instructions, with a lower respective coverage.

At the logic level, parity can be used to protect data words by verifying parity across their individual bits [20]. Parity has been commonly used in commercial processors for the protection of flip-flops in logic components as an efficient alternative to replication, also as illustrated in Chapter 2. At the cell level, various

types of hardening [13] [14] [15] redesign structures of flip flips and SRAM cells to help maintain the correct logic level in the case of an upset. Hardening has also been used commercially in a few designs [44] while roughly doubling the area of protected cells. The previous work has shown the benefit of applying these two hardware resiliency solutions selectively at a fine-grained level when compared to the traditional approach of replication at the architecture and software levels [20] [14]. Aided by flip-flop-level fault injection, this approach has been shown to be more accurate in targeting particular vulnerability at lower overheads. Since these techniques protect against errors in charge that is already stored in a flip flop or an SRAM cell, they are only applicable to radiation induced faults. Techniques such as Razor, not considered here, feature shadow flip flops to specifically protect against timing faults [29].

ABFT is a broad range of application-specific software solutions that take advantage of algorithm properties for error detection. These properties either relate inputs to outputs or allow for checking only computation that makes an essential contribution to the end result in the application. The concept of ABFT was first proposed for matrix multiplication [18], then extended to the Fast Fourier Transform (FFT) [10] and other operations with verifiable computation stages. Due to specific applicability of this approach, most applications have only software-based implementations. Hardware ABFT has only been implemented for FFT but it relies on the redundancy mechanism rather than the checksum [45]. Most of the existing ABFT techniques vary depending on the implementation of the underlying application's algorithm and they have been evaluated on platforms with various vulnerability. Since ABFT for the more sophisticated algorithms can only provide detection capability, the cost of recovery is the ultimate consideration that previously has not been sufficiently evaluated against alternative solutions.

Most previous work determines vulnerability at a coarse-grained level (such as component rather than flip-flop-level) for the purpose of applying and testing of resiliency solutions. Current approaches involve fault injection into application's variables, in software, or into storage components such as the register file and memory, in simulation. Moreover, component-level vulnerability can be also characterized by the Architectural Vulnerability Factor (AVF), which is determined by tracking the occupancy of core components with the amount of sensitive data in simulation [46] or via fault injection. However, these techniques are shown to be inaccurate, since many circuit-level errors are not captured by these models [26]. More accurate flip-flop-level injection techniques are proposed to better reflect the effect of real-world fault rate that can also be emulated with a particle beam. However, such an approach require a fast simulation infrastructure with a flip-flop-level fault injection capability, such as the FPGA-based system that we employ in our experiments [26].

Finally, most previous work evaluates resilience solutions only at a single layer of a system stack (hardware, architecture or software). In spite of extensive previous research, there has been no previous effort to evaluate these solutions together on the same platforms to provide a fair comparison rather than rely on

designer's intuition to chose the most optimal one. Other proposed techniques such as hardware Redundant Multi-Threading (RMT) [35], similar to EDDI, that we evaluate in our work, incur significant performance overheads due to temporal redundancy. Solutions such as monitor cores [30] suffer from noticeable area overhead due to spatial redundancy. Similarly, software symptom-based detection and the corresponding checkpoint-based recovery suffer from long latencies as opposed to localized protection and recovery we evaluate in our work. Control-Flow Checking by Software Signatures (CFCSS) [32], a less capable software version of DFC, and Error Detection by Duplicated Instructions (EDDI) [33], a simple form of RMT, are prohibitive in performance for the level of resilience they provide. We determine these other solutions to be less competitive, for a combination of area, delay and power metrics, and do not analyze them in our work any further. They can be considered as potential design options for optimizing a particular metric.

3.3 Analyzed Protection Techniques

In this section we give an overview of the protection techniques analyzed in this chapter. Our main focus is on architecture-level Data-flow Checking (DFC), the main subject of our study, and the Algorithm-Based Fault Tolerance (ABFT), both of which we implement. We also describe the remaining techniques, including hardening, parity and other, software-level, approaches provided by our Stanford University collaborators.

3.3.1 Architecture: Data-flow Checking (DFC)

Control flow soft errors are caused by faults that affect correct transitioning between basic blocks in the program as well as individual instructions within these blocks. These could include faults in the program counter (PC), branch target or other parts of the core that affects these. The program counter, in turn, depends on the control signals from the decode and exception stage that either hold it or set the next value. The branch target, in the case of an indirect branch, depends on the result of computation in the execute stage and values in the register file. Detection of control flow errors is crucial to the application as they lead to the execution of divergent instruction sequences that are not easily masked and are more likely to affect the program output. Within the context of flow checking, data flow errors relate to faults in the content of processed instructions. These can be caused by faults in the instruction word that occur in memory or during processing in the pipeline stages before the checker. Although faults in the instruction word are more likely to be masked, they can cause execution of an incorrect operation and change values of data operands which, in turn, can also cause control-flow errors and affect output of the program.

The data-flow checking (DFC) approach [16] [17] protects against errors in the flow of basic blocks and individual instructions in the program. It does this by statically embedding a control flow signature,

for each basic block, in the binary and then dynamically verifying it at run time. This functionality is achieved with the following steps. The control flow graph in the binary is extracted by the compiler to find basic blocks that are delimited by branch instructions. For each of the two targets of a branch instruction (fall-through and jump) a signature is statically generated based on content (program counter, opcodes and operands) of individual instructions in the corresponding target basic block (Fig. 3.1). The signature is then embedded in an additional instruction slot or alternatively in a repurposed branch delay slot (MIPS or SPARC architectures). At run time, the hardware DFC checker mechanism dynamically reconstructs the signature and verifies the instruction control flow by comparing the dynamic signature to the one embedded in the binary. This check takes place when a block-delimiting branch instruction is encountered in the pipeline stage where the checker is placed. Finally, the recovery for DFC requires storing register values in a shadow file as well as buffering memory writes until the check is performed.

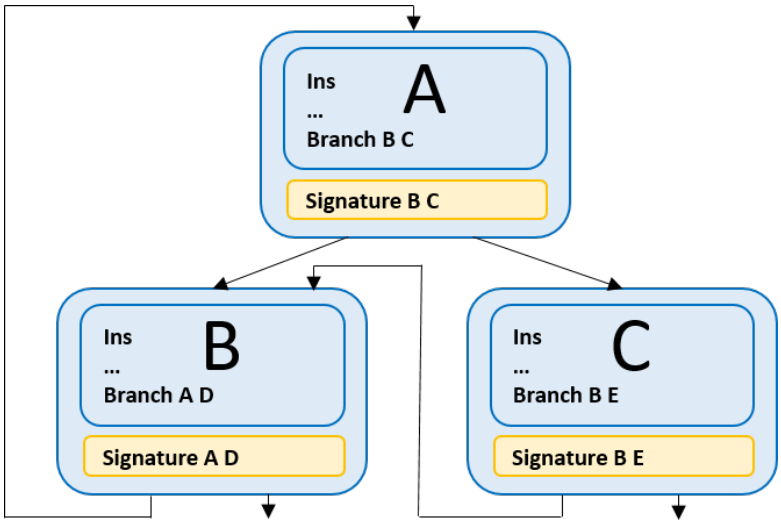

Figure 3.1: Data-flow checking (DFC): Valid transitions between basic blocks in the binary on branch instructions. Signature instructions include static data flow information used for verification.

By design, DFC is capable of detecting control flow errors that would show in the sequence of instructions that it monitors. Since these errors can originate in various core components that are involved in maintaining the correct flow, DFC provides some degree of protection in those relevant circuits. However, since not all of the faults in these core components translate to control flow errors detectable by DFC, their protection is incomplete. Moreover, most of the operations performed in the pipeline rely on control signals derived from the opcodes and operands in the instruction word, rather then the original instruction word. Therefore checking for the effects of corruption in this content would require verification of these control signals against the original instruction word in the signature. However, the main advantage of the DFC technique is that it is able to compress and check a large amount of control state which would otherwise require more comprehensive

protection with traditional techniques. Therefore, within its respective coverage scope, it is a potentially competitive alternative for low-overhead hardware solutions such as hardening and parity. The complete assessment of benefits also requires evaluation of the recovery mechanism.

Due to DFC design trade-offs and differences in the underlying core architectures, the scope of DFC varies across implementations. The differences are related to the types of instructions protected (all, PC-content only, branch instructions only), signature generation algorithm and the resulting coverage of the flow related features in the core. Although not presented here, we also implement and analyze two other, less capable, versions of DFC, namely control-flow checking (CFC). In the first, equivalent to some previous work and the software implementation [32], the coverage is limited only to branch instructions responsible for transitioning between basic blocks. In the second, equivalent to the implementation from the University of Texas at Austin, only the program counters of instructions are included in signatures and verified at run time. Moreover, the development of DFC involves challenges related to compiler and hardware modifications as well as the non-trivial cost of recover, all of which have limited its implementations in the past.

3.3.2 Hardware: Hardening and Parity

In Chapter 2, among the hardware-level solutions, we determined both hardening and parity to be efficient solutions for the protection of logic components in the core, hence we evaluate them using implementations provided by our collaborators from Stanford University. Hardening provides soft-error protection for storage circuits (SRAM) and sequential logic (flip-flips). In our study, we use it for the latter. The approach involves adding redundant transistors to storage cells to provide redundancy that reinforces the logic state [13] [14]. Other variations implement a feedback path that cancels the effect of an upset [15]. There are a range of hardening variants that offer trade-offs in area, power and delay versus the amount of resilience provided (Table 3.1). To achieve the best possible soft-error-rate (SER) rate, we use the heavy version [14] in our analysis, which roughly doubles the area of the protected cell. This solution is implemented at the library cell level with no changes to the logic design. The intrinsic error cancellation capability eliminates the need for a recovery solution. Although the overhead of hardening is comparable to that of replication at the cell level, it allows selective application to only vulnerable cells to reduce the overall core-level overhead.

Table 3.1: Different grades of flip-flop hardening.

Type	SER	Area	Power	Delay
Baseline	1	1.0	1.0	1.0
Light	2.5x10-1	1.2	1.1	1.2
Moderate	5.0x10-2	1.3	1.5	1.7
heavy	2.0x10-4	2.0	1.8	1.0

Parity provides soft-error protection for storage circuits (SRAM) and sequential logic (flip-flops) [20]. In our study, we use it for the latter. A parity encoder generates a parity signal based on the inputs to a group of flip-flops. When the flip-flops outputs are accessed, the signal is compared against the one generated by a decoder circuit based on the outputs. The error signals from different parity groups are then aggregated into a single error output (Fig. 3.2). Due to the long XOR-tree implementation, this solution suffers from the increased delay that may necessitate more expensive pipelining to preserve the original design frequency. Although, parity incurs a lower area overhead than hardening, it has a lower resilience capability per protected bit due to a single parity bit used for detection. Moreover, consideration of several design parameters such as parity group size, flip-flop vulnerability, cell locality, timing slack and recovery are required to arrive at an optimal design. For a lower resiliency target, parity can be selectively applied in combinations with hardening, where possible, to lower the overall core-level area overhead. In the cross-layer solutions involving DFC and ABFT that we analyze in this chapter, we use hardening and parity for the protection of the remaining circuitry to reach desired soft-error-rate (SER) target.

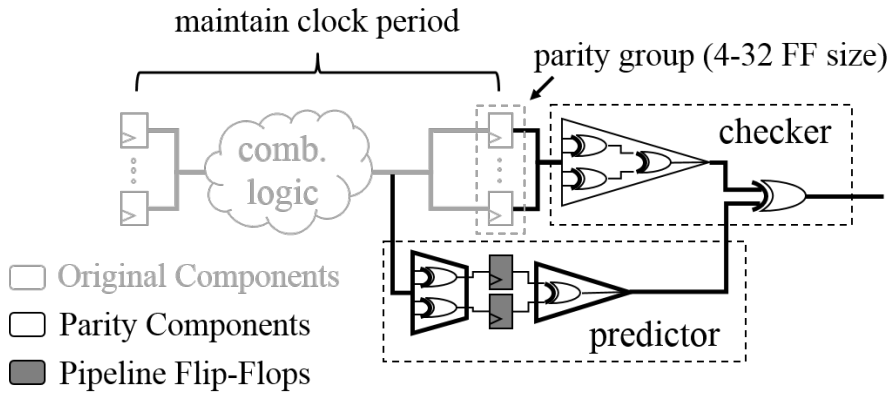

Figure 3.2: Logic parity: Latched data is verified against parity bit computed in the previous cycle based on data. Pipelining is necessary to avoid increasing critical path and frequency.

3.3.3 Software: Algorithm-based Fault Tolerance (ABFT) and others

Among the considered software-level solutions, Algorithm-based Fault Tolerance (ABFT), protects programs by taking advantage of specific properties in a particular algorithm. These properties could include relationships between inputs and outputs or checking only computation that makes an essential contribution to the program output (Fig. 3.3). Similarly to other specific solutions such as arithmetic codes or DFC, this technique has a potential to efficiently provide coverage that would otherwise require more comprehensive protection with traditional techniques. However, unlike the former techniques, ABFT is only applicable to a particular applications which necessitates additional protection for other workloads. However, ABFT might eliminate

the need for other protection in the context of an accelerator, or an application-specific integrated circuit (ASIC). Since this techniques only protect computation, it cannot detect errors during I/O activity that can account for a significant part of the program execution. Moreover, in the context of a general-purpose processor, many faults that take place in the core, the operating system code and the application are not detectable by ABFT but can affect the final program output. Although ABFT does not incur area and power overhead from additional hardware, it suffers from performance overhead from additional checking computation. For applications that consist of elementary operations, such as matrix multiplication [18], ABFT can provide both detection and correction. More complex operations involving a series of custom transformations, such as FFT [10], only allow for error detection while correction is achieved via retry at an increased performance overhead. The detection capability of ABFT and vulnerability of the underlying hardware differ in the case of accelerators and ASICs as shown in Chapter 4.

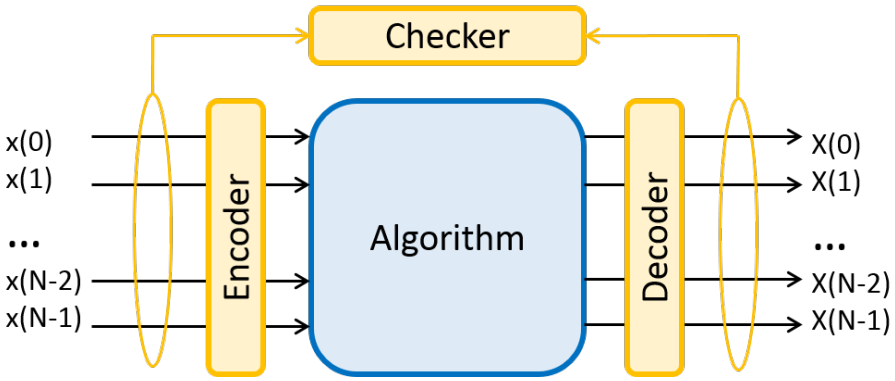

Figure 3.3: ABFT for FFT application. Sum of output points is compared to the first input point.

As a part of our cross-layer analysis we also consider Control-Flow Checking by Software Signatures (CFCSS) [32] and Error Detection by Duplicated Instructions (EDDI) [33] protection techniques analyzed by our collaborators from Stanford University. Similarly to ABFT, these software-level solutions incur no overhead due to additional hardware, but unlike ABFT, they are generic and applicable to all workloads. CFCSS is a software version of the least capable version of DFC, where only transitions between basic blocks on branch instructions are verified. This technique adds unique signatures to each basic block, at the level of the assembly code during compilation, which are then used to verify the correct flow during execution. Because of a typically small size of a basic block (4-5 instructions), this solution suffers from performance overhead due to multiple additional instructions per block.

During compilation, EDDI replicates each instruction to compare the outputs during execution to ensure correct operation. As a result, it incurs performance overhead due to additional redundant and compare instructions. EDDI is similar to Redundant Multi-Threading (RMT) [35] which allows for partial amortization

of the overhead in a superscalar core. Although not evaluated here due to the unfinished implementation, the optimal version of this solution is its selective application only to the most vulnerable instructions. We also developed a prototype hardware EDDI implementation, for evaluating its improved performance benefits and hardware overhead. The coverage considerations of CFSS and EDDI related to the underlying architectures and the software stack are similar to those for ABFT.

3.4 Platforms

In this section we give an overview of Leon and IVM, platforms used for experiments in this chapter. These platforms are representative proxies for the simple and more complex cores, respectively, that allow us to see the impact of the analyzed solutions and corresponding design choices for the two architectures.

3.4.1 Leon

Leon3 is a simple in-order (IO) reduced-instruction-set-computing (RISC) processor obtained from Gaisler Research [47]. It has been used in multiple research and commercial implementations. The processor features a scalar 7-stage pipeline that includes the following stages: fetch, decode, register access, execute, memory, exception and writeback. It implements a 32-bit Sparc V8 instruction set architecture (ISA) with a minimum of 2 register windows. Our version features hardware integer and software floating-point instructions. We use a typical configuration with 4kB instruction and data caches and a 2-window 32-entry 32-bit register file. The optional FPU, TLB and SOC components are not used in our setup. Using the 28nm library and sweeping in increments of 50MHz, the maximum post-layout design frequency of this core was determined to be 2GHz. The area distribution across core components shows that memory units (caches and the register file) dominate over logic of of the simple in-order core (Fig. 3.4a). Execute and exception pipeline stages have the largest numbers of flip flops due to more complex arithmetic and control/flush mechanism there (Fig. 3.4b). The instruction and program counter data are generated in the fetch stage and forwarded throughout the pipeline. Various control signals generated based on these inputs and forwarded to other pipeline stages. The register file lies in the critical path of the core and determines its frequency.

3.4.2 Illinois Verilog Model (IVM)

Illinois Verilog Model (IVM) 1.0/1.1 is a more complex 32-bit out-of-order (OO) RISC processor obtained from the University of Illinois [48]. The processor features a superscalar 12-stage pipeline with 8-wide fetch, 4-wide decode, 4-wide rename, 6-wide issue, execute, memory and 8-wide retire stages as well as speculative

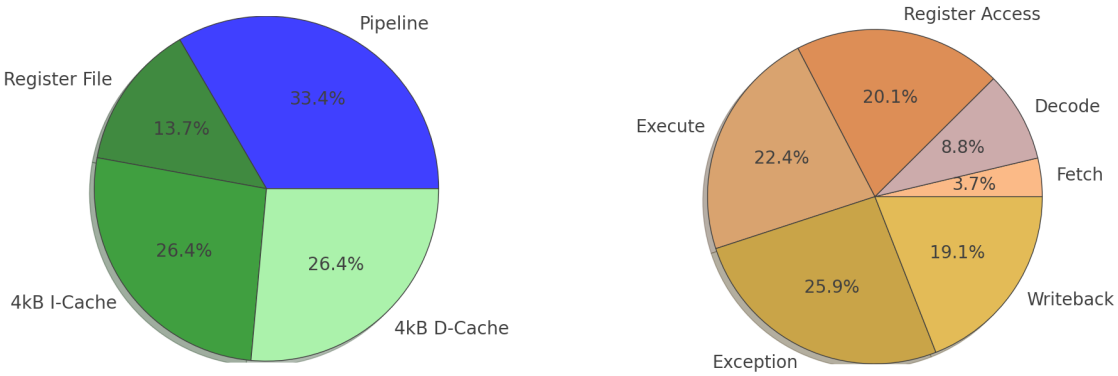

Figure 3.4: Leon core: a) area distribution across core components; b) flip-flip distribution in the pipeline. Pipeline is the largest component while exception stage contains most flip-flops.

execution and a branch predictor. It implements a 64-bit Alpha ISA similar to Alpha 21264 and AMD Athlon. Similarly to Leon, our version features hardware integer and software floating-point instructions. We use a default configuration with 8kB instruction cache, 32kB data cache and an 80-entry 64-bit register file. Using our 28nm library and sweeping in increments of 50MHz, the maximum post-layout design frequency of this core was determined to be 600MHz. The frequency is lower than expected because the design was not optimized with an intention of full synthesis. The area distribution across components shows that the complex OO core dominates over memory units (caches and the register file). This is also because the pipeline includes a noticeable amount of multi-port storage arrays (Fig. 3.5a). The register file is also relatively large due to the multi-ported implementation. Components related to OO execution make a significant flip flop contribution to the core (Fig. 3.5b). The instruction word and program counter, which are generated in the fetch stage, are forwarded till the last unit in the stage, which is somewhat uncommon for the OO implementations. Among others, these signals are also used to generate control signals in each pipeline stage. The retire stage also includes exception logic, which is in the critical path of the core.

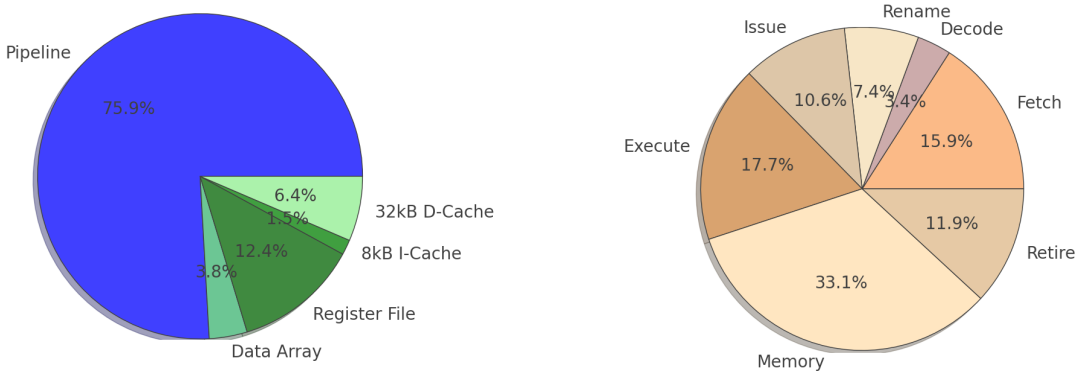

Figure 3.5: IVM core: a) area distribution across core components; b) flip-flip distribution in the pipeline. Pipeline is the largest component while memory stage contains most flip-flops.

3.5 Implementation

In this section we elaborate on the design details of Data-flow Checking (DFC) in Leon and IVM architectures, the main protection technique that we implemented for the analysis in this chapter. More details of the Algorithm-based Fault Tolerance (ABFT) approach for the FFT application, that we also implement, are given in the next chapter where we evaluate it for the accelerator architecture. Moreover, we also provide an overview of considered recovery solutions including the ones that we design for DFC.

3.5.1 DFC in Leon

We first modify the Leon compiler to ensure that all delay slots, available in the SPARC instruction set architecture (ISA), are preempted and filled with NOP instructions to make space for embedding of signatures. Some of these slots would otherwise be filled with useful instructions, where possible. Our approach manages to empty delay slots in most of the binary that includes the user code, libraries and other (error injection) code. We determined that the alternative method of inserting empty instruction slots via assembly padding was unfeasible for a large Leon C library distribution. The lack of the ability to insert signatures in the library would result in confining signatures to the user code in the binary thus, making coverage analysis more difficult.

We extended an existing binary parsing tool [49] to include decoding for the SPARC instruction content. This functionality would normally become part of the compiler in a commercial implementation. On top of this feature, we build a signature algorithm that extracts control flow graph from the binary and identifies valid branches with the corresponding target blocks. The algorithm then generates signatures for the basic blocks and embeds them in the empty delay slots. Signatures are generated only for both conditional and unconditional direct branches (where targets can be determined statically) that have empty delay slots and targets that are valid within the scope of the binary. Basic blocks originating from indirect branches are not covered by this solution since their targets cannot be statically determined at compile time.

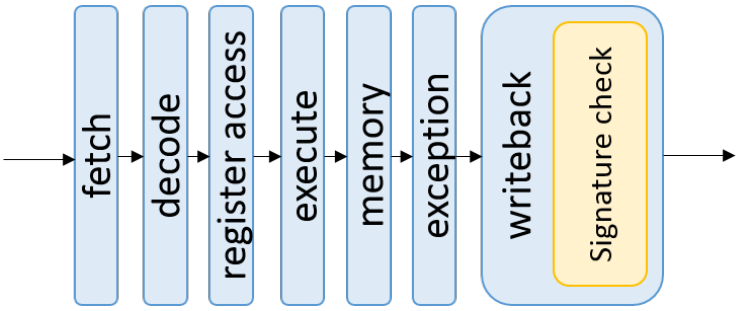

Figure 3.6: Leon core: Data flow is verified by DFC checker in the writeback pipeline stage by comparing static signature to the dynamic data-flow representation constructed during execution.

The signature is generated based on the program counter (PC), opcodes and operands of individual instructions in the basic block. In order to minimize aliasing, the signature uses all 22 bits (11 bits for each of the two branch targets) available in the instruction slot. The binary tool and the hardware checker employ a hashing function that performs an XOR operation on the most diverse bits (statistically determined) from the instruction content in way that improves signature distribution and minimizes aliasing. We determine that, while confined to a fixed number of bits, more sophisticated polynomial-based hashing implementations are expensive in terms of hardware and provide little improvement in terms of aliasing. For the fixed size RISC instructions in the SPARC ISA, we also implement 7-bit signatures that could be embedded in unused bits of other instructions in the basic block, where possible, to lower impact on performance.

The Leon design was restructured to separate pipeline stages and relevant registers for easier integration with the DFC checker and to obtain per-component results. The design of the checker consists of the decoder for the relevant control-flow instructions (branches, delay slots), the state machine that collects instruction data and handles flow irregularities (multi-cycle instructions, traps, interrupts and exceptions), as well as the signature unit that constructs a dynamic run-time signature and compares it to the static one from the binary. Since instruction information is forwarded all the way to the writeback stage, the checker was placed there to cover accumulated errors from the previous stages (Fig. 3.6). In order to increase the coverage of the execution flow, we also included checking for control signals to arithmetic units and the load-store unit which can be derived from and verified against instruction data in the signature. Finally, structures for the error injection framework were also added to the core (described later).

Our implementation of the checker varies from Argus [17]. This work features three types of signatures (for the PC, registers and memory address) evaluated locally and then combined into one when the basic block is checked. The signatures for registers are calculated based on the operand registers' signatures and the opcode of the instruction that stores data in the register. We originally implemented this approach but it yielded no detection benefits over the one we use while complicating the checker and adding overhead due to a larger number of signatures generated. While separate signatures do not provide much benefit due to small basic block size, they lose their resolution when merged into a single 5-bit signature. However, with the way signatures are forwarded through core units, Argus provides additional coverage for control signals generated in the decode stage and used in the execute stage. To compensate for this, we add a checker that verifies correctness of operands and the majority of control signals by deriving them and comparing against the original instruction that we use for the signature.

3.5.2 DFC in IVM

Based on our experience and available resources, we determined that developing a DFC infrastructure for IVM, equivalent to that for Leon, is not feasible due to multiple reasons. First, the cross-compiler for the particular Alpha ISA used by IVM is not available, nor have we access to the native compiler in any machine with an Alpha processor. The IVM core was released with an assumption that precombiled binaries could be used. As a result, we are not able to embed DFC signatures in the binary via modifications to the compiler. Moreover, the Alpha ISA does not feature delay slots such as those present in the Sparc ISA used by Leon. This would necessitate manual modification of a large number of library files to insert an additional padding compilation step for the embedding of signatures. While this is not a feasible task, skipping this step, on the other hand, would confine signatures and corresponding coverage only to the user code. Moreover, similarly to Leon, the IVM core experiences flow irregularities due to multi-cycle instructions, traps, exceptions and interrupts, that can all affect the DFC checker. Based on our experience with Leon, we determined that it would take a significant amount of time to gain sufficient understanding of the IVM implementation details to compensate for these events in the checker.

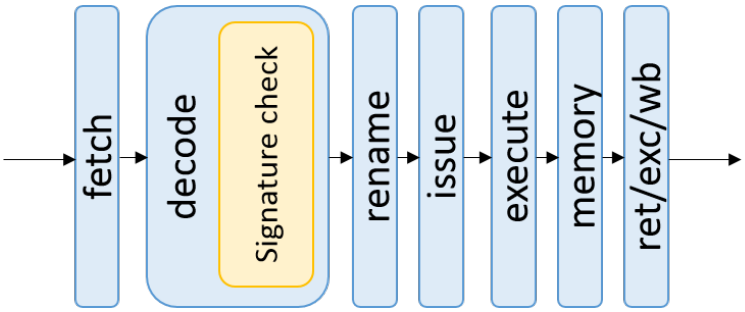

Figure 3.7: IVM core: DFC checker is placed in the decode stage because complete instruction and program counter data are not available beyond that stage is the out-of-order (OO) architecture.

Therefore, for the IVM core, we take a different approach, in which we detect errors in signals relevant to the signature by comparing against the error-free run, rather than utilizing a typical checker mechanism. To ensure that this method is equivalent to the checker implementation, we first determine the program counter and instruction information that would normally be used by the checker for the generation of the signature. Although the IVM implementation forwards these signals from the fetch till the retire stage, we place the checker in the decode stage (Fig. 3.7). This is the last stage where these signals are still available in a typical OO core (before the renaming of operands takes place). We then use a combination of architecture and RTL simulation to run the benchmark in the processor core with error injection while storing values of these signals into the log at every cycle. The same is done for the error-free simulation that runs in parallel. Both logs are then compared to detect errors in the monitored signals at the bit-level or in the data flow any time

during benchmark execution. Care is taken to detect only the first error that would trigger recovery and not the subsequent ones that are a consequence of the first. Since this method analyzes the original data, it assumes that the signature mechanism, present in the full implementation, would compress the relevant signal information and convey it with little or no loss until the check is performed. As we explain in the following sections, this is true for most practical purposes when using a relatively accurate hashing function such as that used in our Leon implementation. For the purpose of estimating the area of the checker, we assumed a design similar to that in Leon, since roughly the same functionality and amount of processing are involved.

3.5.3 Recovery

The choice of recovery solution varies, depending on the detection latency of the protection mechanism and the amount of acceptable roll-back in the execution process. The simplest mechanism utilizes the operating system to squash the entire workload on a detected error and restart its execution while suffering a loss of the results obtained so far. The most popular mechanism, on the other hand, relies on taking periodic software-level checkpoints of the application state that can be used to roll back on a detected error with partial loss of computation results. Since both of these suffer from a long recovery latency, we only consider more localized and efficient solutions in this chapter. Moreover, the more frequent restart and checkpoint recovery might not be tolerable in operating environments with a higher error rate (at high altitudes) where it would disrupt equipment functionality. The simplest local recovery solution takes advantage of the built-in recovery mechanism in the core that can flush any erroneous state from the pipeline if the error is detected before reaching the memory stage (Fig. 3.8). This solution is applicable to parity due to its relatively short detection latency. However, this latency is longer in the pipelined version, which restricts the use of parity beyond the execute stage. Another solution includes a recovery unit, similar to the IBM R-Unit [5], which buffers processor state and writes to the register file a duration equal to the pipeline length, thus allowing full recovery of the pipeline (Fig. 3.9). We evaluate the overhead of these two local recovery solutions by analyzing the corresponding hardware structures that include buffers and additional register files.

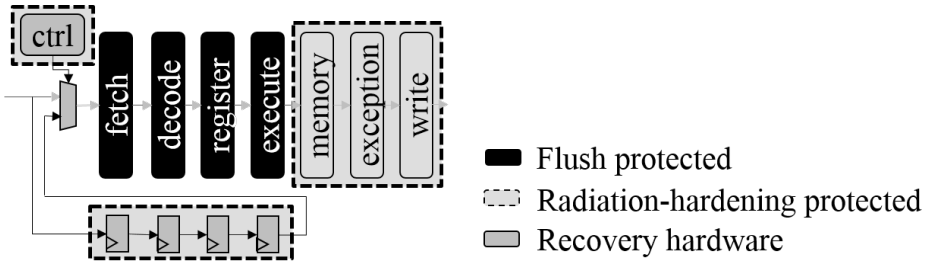

Figure 3.8: (Flush) recovery unit for parity: Architecture state can be restored and registers can remain unmodified if error is caught before reaching memory stage.

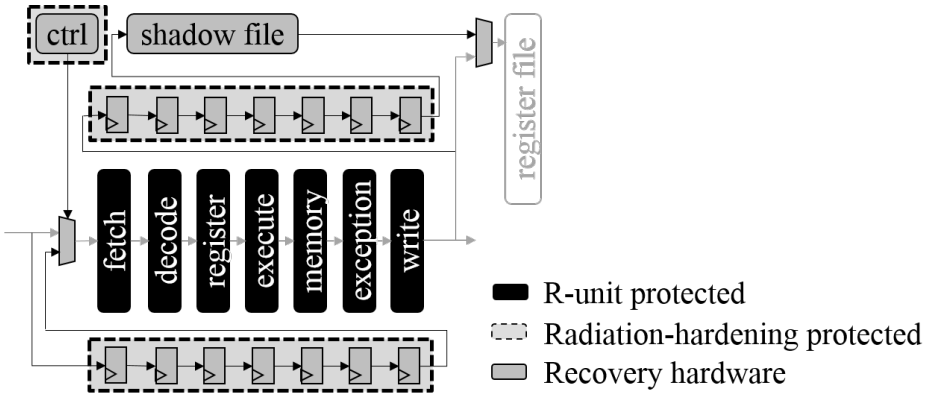

Figure 3.9: Recovery unit for parity: Architecture state and register file can be restored from redundant registers to 7 cycles before, enough to cover pipeline length.

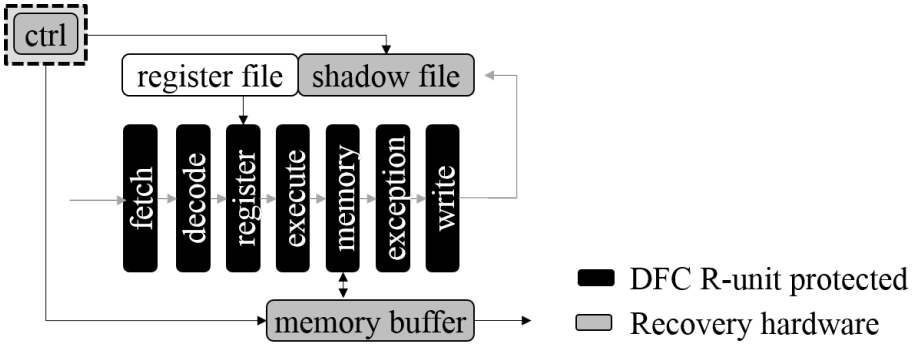

Figure 3.10: Recovery unit for DFC: Due to variable basic block size, a memory write buffer and shadow register file (only Leon core) are required for recovery.

The corresponding local recovery solution for DFC is more complex than for parity due to the unbounded detection latency of DFC. The core checkpoints the architecture state and the register file at the beginning of every basic block, as well as buffers all memory writes during the execution of the block. In the case of the Leon in-order (IO) architecture, this data is stored in a corresponding latch array, shadow register file, and a content-addressable memory (CAM) buffer structure, respectively (Fig. 3.10). The memory buffer is sized according to the largest anticipated number of writes in a basic block (around 15). When a flow error is signaled by the checker, the memory write requests are discarded while the architecture state and register file are restored. We evaluate the cost of this solution, mainly contributed by the controller circuitry and the memory buffer. In the IVM out-of-order (OO) core, the shadow register file and the memory buffer are unnecessary, as the outstanding register and memory writes can be held in the re-order-buffer (ROB) until the check is performed while significantly lowering the area overhead of this solution. Moreover, the buffering capability of ROB also eliminates the need for buffering in the recovery unit used for parity while also lowering the overall overhead.

3.6 Methodology

In this section we explain our methodology for the study. Prior work on fault tolerance has typically proposed techniques that target a single layer of the system stack (hardware, architecture, software) and made projections regarding its benefit over others. A fundamental difference of our methodology is that we apply a systematic approach to provide a fair evaluation of multiple efficient resilience choices and their cross layer combinations rather than leave it to designer intuition to chose between those and to determine their optimal applicability.

3.6.1 Fault Model

In this chapter, as well as throughout the dissertation, we focus on radiation-induced transient faults, particularly single event upsets (SEUs), as a vehicle to study soft errors. This allows us to consider a wider set of protections techniques including those, such as hardening and parity, that are only applicable to radiation-induced faults as opposed to timing violations on voltage droops. Moreover, the transient nature of this type of fault allows us to explore a wider range of design trade-offs due to masking effects at different layers of the system stack. We specifically focus on sequential logic, since there already exists many well-established and inexpensive techniques for protecting regular, array-type, structures like memories. Furthermore, we do not specifically address errors in combinational logic, since studies indicate it is far less (one or two orders of magnitude, depending on technology) susceptible to soft errors compared to sequential logic [3].

3.6.2 Fault Injection

We use fault injection to simulate soft errors due to SEUs at the flip-flop level [26]. The framework injects a single fault into a random flip-flop at a random cycle during a single benchmark run. For the Leon platform, the injection is performed during the benchmark run in the core implemented in a Field Programmable Gate Array (FPGA) chip, the infrastructure provided by our collaborators from Stanford University. This setup provides a significant simulation speed-up over the RTL simulation in a general-purpose processor for an individual run. We have several of the FPGA chips available in the BEE (Berkeley Emulation Engine) boards [50] to allow for multiple parallel runs. For the IVM platform, the injection is performed during the benchmark run in the RTL code simulated in the Stampede supercomputer [51]. The availability of the supercomputer allows for a large number of concurrent runs, which reduces the overall run time for a batch of simulations. The injection frameworks for both Leon and IVM required integrating additional RTL code with the core to connect to the flip-flop network. Data presented for our analysis is based on at least 10,000

and 150,000 injections in each of the benchmarks used, for Leon and IVM, respectively. This allows each flip-flip to be injected at least 10 times at random instances during benchmark execution.

3.6.3 Flip-flop Ranking for Hardening/Parity

For the application of hardening and parity, flip-flops are ranked to reflect the likelihood to propagate faults [26]. This is done via the accurate flip-flop-level injection methodology provided by Stanford University that allows determining vulnerability of each flip-flop and its contribution to meaningful errors. The approach allows us to use these techniques efficiently by applying selectively, starting from the most vulnerable flip-flops, to cover parts of cover circuitry for a given SER rate. Since intrinsic coverage of hardening, a cell-level technique, cannot be evaluated in RTL-level injected simulation, it is assumed based on previously conducted SPICE experiments [13].

3.6.4 Error Metrics

The outcome of the injected run is analyzed according to five categories [26]: vanished, output not affected (ONA), output mis-match (OMM), unexpected termination (UT), and hang to describe its effect on the core state and the visibility to the user. An error has vanished when there is no effect on the core state or the program output values. The ONA error takes place when there is an effect on the core state, but not the program output values. Finally, meaningful errors are categorized as OMM, UT or hang if they have an effect on the output values, or cause the program to crash or hang, respectively. Under these three categories, the core state could also be affected. For each flip-flop we calculate the vulnerability metric by dividing the number of occurred/unprotected errors by the total number of injections. Since taking vanished errors into account (Eq. 4.1), is not very useful, we focus on two other metrics to reflect the all present (non-vanished) and meaningful (visible to the user) errors rates, respectively. The first takes into account ONA, OMM, UT and hang categories (Eq. 4.2), while the second only OMM, UT and hang (Eq. 4.3). The vulnerability, or the Soft Error Rate (SER), improvement is calculated by dividing the vulnerability of the unprotected design by that of the protected design (Eq. 4.4). Moreover, when analyzing coverage in each of the considered architectures, we use two metrics to describe its different aspects: touch range and detection rate. Touch range includes all flip-flops where the protection technique detects at least one error at some instance of time, while detection rate is the number of detected errors divided by the number of occurred/meaningful errors.

$$\text{all-error vulnerability} = \frac{\text{vanished, ONA, OMM, UT, hang}}{\text{all errors}} \tag{3.1}$$

$$\text{meaningless-error vulnerability} = \frac{\text{vanished, ONA}}{\text{all errors}} \tag{3.2}$$

$$\text{meaningful-error vulnerability} = \frac{\text{OMM, UT, hang}}{\text{all errors}} \tag{3.3}$$

$$\text{vulnerability improvement (a.k.a. SER improvement)} = \frac{\text{unprotected vulnerability}}{\text{protected vulnerability}} \tag{3.4}$$

3.6.5 Collaboration with Stanford University

Our collaboration with the Stanford University was a very interactive process. The Data Flow Checking (DFC) and the corresponding infrastructure (compiler changes, binary tools, hardware design) were implemented at UVA. Algorithm-Based Fault Tolerance (ABFT) for four of the benchmarks (four versions of FFT) was also implemented at UVA. The Stanford University provided the choice of platforms and the fault injection framework. They also performed analysis of hardening/parity for augmentation of other techniques as well as analysis of other ABFT benchmarks, Control-Flow Checking with Software Signatures (CFCSS) and Error Detection with Duplicated Instructions (EDDI). The coverage analysis of DFC and a set of ABFT benchmarks was performed at UVA by using modified fault injection framework and multiple analysis tools developed there. Coverage data was forwarded to Stanford University for augmentation with hardening/parity.

3.6.6 Other Infrastructure

Analysis is conducted on a diverse set of benchmarks including the SPECINT 2000 benchmark suite [40] and the DARPA PERFECT benchmark suite. Our results are presented for the latter since most of the hardware and architecture-level cross-layer work was performed with this suite. The PERFECT suite includes benchmarks with various compute and memory characteristics designed to represent workloads in military equipment. They include time-frequency transforms, space-time processing, radar, imaging and sorting. The BCC compiler [47] (a GCC wrapper) was used to compile benchmarks for Leon. Precompiled binaries provided with IVM were used with this architecture. The BCC compiler and the ELF binary tool [49] were modified for the Leon DFC implementation. Physical design overheads are evaluated using a 28nm TSMC cell library [52] and Cacti [42]. Synopsys design tools (Design Compiler, IC compiler, and PrimeTime) [53] are used to perform RTL simulation, synthesis and place-and-route as well as area, power and timing analysis.

3.7 DFC Results

In this section we present area, power, performance and coverage characteristics of the DFC protection technique applied alone to Leon and IVM architectures.

3.7.1 Area and Power

In the Leon architecture, the area contribution of the DFC checker is 6.6% of the core pipeline and 3.2% of the core with memories included (Fig. 3.11a). This area includes circuitry used for instruction decoding, collecting of instruction data, handling flow irregularities and generating signature. The latter contributes the least overhead to the core. The corresponding power consumption is 10.1% and 4.5% for the pipeline and the entire core, respectively (Fig. 3.12a). These numbers are higher than the area overhead with respect to other components since the checker circuitry has a high utilization for any given instruction encountered.

In the IVM architecture, the estimated area contribution of the DFC checker is about 0.16% of the core pipeline and 0.15% of the core with memories included. This area corresponds to roughly the same checker design as in Leon. However, the resulting overheads are negligible (and therefore not shown in figures here) because the out-of-order IVM design is close to 50 times larger than Leon. The power consumption is 0.20% and 0.19% for the pipeline and the entire core, respectively. The actual Leon and IVM designs that are tested also include additional circuitry for error injection which we did not consider here for clarity.

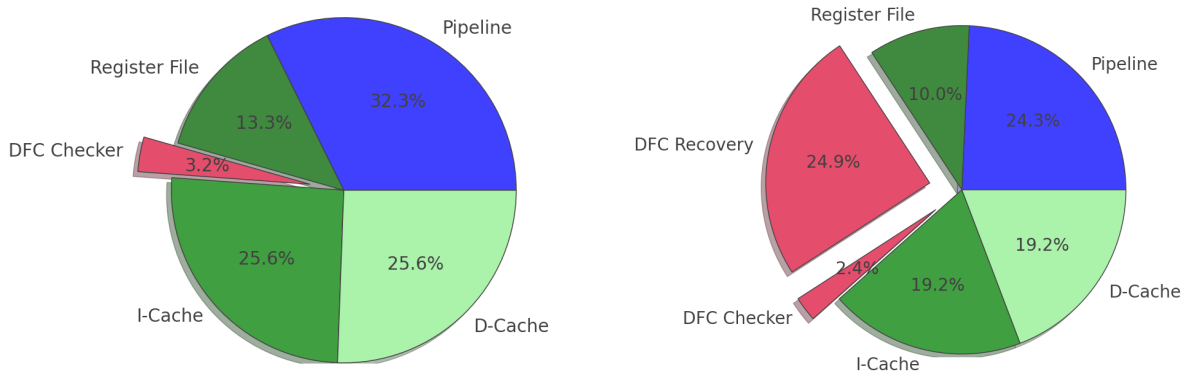

Figure 3.11: Leon core: a) Area overhead of the DFC checker; b) Area overhead of the DFC checker with recovery. Recovery solution for DFC adds significant overhead to simple IO Leon core.

The area contribution of the DFC recovery unit in the Leon architecture is the chief cost: 40.4% of the core pipeline and 24.9% (27.3% with the checker) of the core with memories included (Fig. 3.11b). This is significant in the context of the small size of the Leon3 in-order (IO) core . This area is mostly contributed by the replication of the register file, which alone accounts for 10.0% of the core. The second largest contribution is the buffer for memory write requests, which is sized according to the size of the largest basic block (10.4%).

The control circuitry for the architecture state checkpointing, reading/writing from/to buffer and memory accounts for the rest of the overhead (4.5%). The corresponding power contribution is 45.31% and 26.65% (29.9% with the checker), for the pipeline and the entire core, is relatively high due to utilization of the control circuitry there (Fig. 3.12b).

In the case of the IVM architecture, the estimated area contribution of the DFC recovery unit is 0.15% of the core pipeline and 0.14% of the core with memories included. These overheads are contributed only by the circuitry for the architecture state checkpointing since the re-order buffer (ROB) in the IVM architecture can be used for buffering of register and memory write requests while eliminating the need for additional storage. These overheads are also negligible in the context of the overall core area. The power consumption is 0.17% and 0.15% for the pipeline and the entire core. Similarly to Leon, this is relatively higher than the rest of the core logic, per unit area, due to due to similar utilization to that seen in the Leon core.

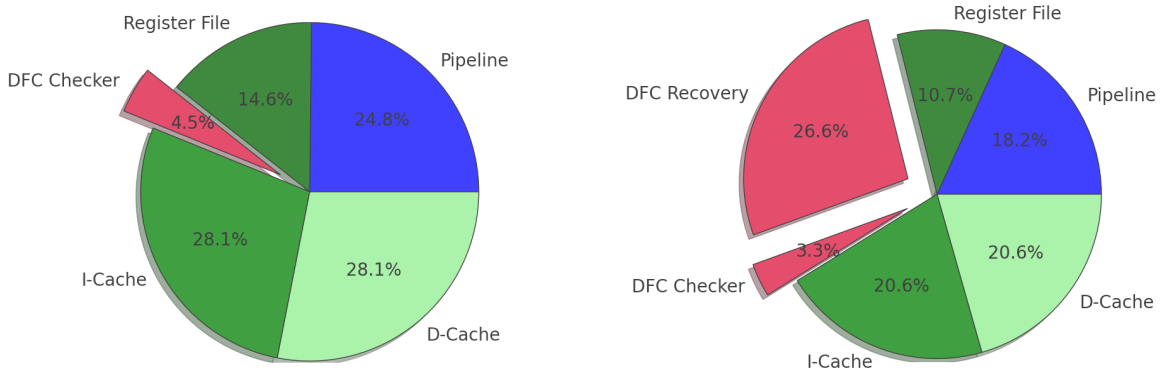

Figure 3.12: Leon core: a) Power overhead of the DFC checker; b) Power overhead of the DFC checker with recovery. Recovery solution for DFC adds significant overhead to simple IO Leon core.

As mentioned earlier, we also evaluate two less capable versions of DFC for the Leon core, which are variants of Control-Flow-Checking (CFC). For both of these, the overheads of the checker and the recovery solution are almost the same as those for DFC, which removes potential for area optimization and trade-offs. This is explained by the fact that the checker is dominated by instruction decoding and data flow state machine rather than the signature generator. The area of the recovery solution is the same, regardless of the type of signature generation and coverage of instructions within the basic block.

3.7.2 Performance

In both Leon and IVM architectures, checking is performed in parallel with the processing in the stages where checkers are placed (writeback and decode stage, respectively). In the Leon core, the delay of the checker circuitry is about 24.7% of the register file in the register access stage, the slowest part of the core. This delay is even less significant for the IVM core that runs at more than 3 times lower frequency, where the

retire stage is in the critical path. Therefore the DFC checker does not affect the critical paths delays or operating frequencies in the two architectures.

The performance overhead of DFC in terms of computation cycles depends on the sizes of basic blocks that execute, where each requires an additional instruction to hold the signature. Therefore, the overhead is higher for smaller basic blocks. Figure 3.13 shows how these sizes vary across PERFECT benchmarks (8.19 cycles, on average) resulting in varied performance overhead. Since the branch instruction is not resolved until 3 cycles after the fetch, it is followed by the delay slot (sometimes filled with an instruction from the jump path) and two speculative cycles from the fall-through path. Depending on the branch outcome, the remaining instructions are either the remainder of the fall-through path or the entire jump path, including additional cycles due to memory stalls and multi-cycle instructions. Since most branches, such as those in loops, are taken, it is usually the latter.

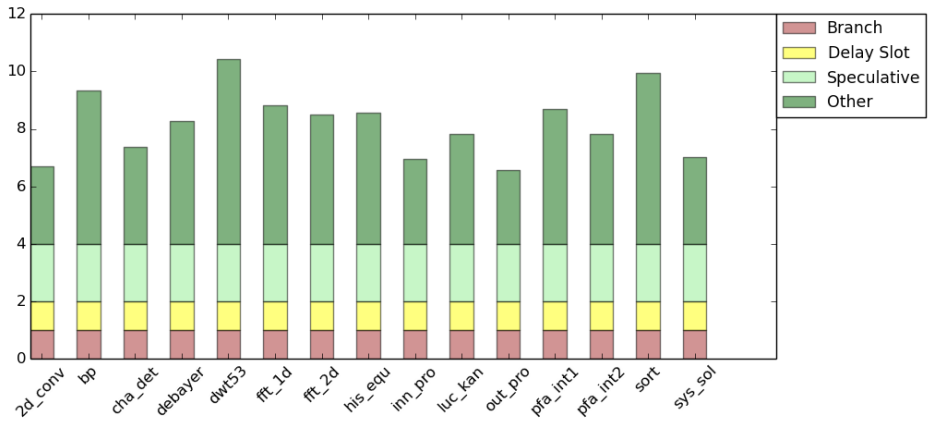

Figure 3.13: Leon core: Dynamic instruction count in the basic block, for PERFECT benchmarks. Relatively small size of basic blocks causes additional signature instructions to incur more significant performance overhead.

Since the SPARC ISA used in the Leon architecture allows preempting its delay slots, those can be leveraged for the purpose of embedding signatures. The performance overhead (9.2% on average) is then determined by additional instructions slots that are created to hold the useful instructions removed from the preempted delay slots (Fig. 3.14). Most other architectures, such as IVM, that do not feature MIPS or SPARC ISA, require an additional instruction for each basic block to hold the signature. Although presented for the Leon core, this overhead is also visible in Fig. 3.14. This would result in an overhead of around 10.8%, on average. These overheads are similar because preemption (of mostly useful instructions) requires creations of a similar number of additional instructions.

In the case of both architectures, the overhead can be reduced by embedding signature instructions in unused bits of instructions in the basic block, where possible. We determine that 57.2% of basic blocks have

2 or more instructions that are of the correct format with the unused intermediate value field that allow embedding 2 parts of the signature instruction. For both Leon and IVM architecture, this allows lowering the performance overhead of signature instructions to 4.7% and 6.3%, respectively.

To put these overheads into perspective, it should be noted that the introduction of an additional signature-related instruction adds only one cycle to the processing of the basic blocks in the dynamic instruction count. However, in the in-order (IO) Leon core, 32.8% of cycles in a basic block, on average, are due to memory stalls and execution of multi-cycle instructions. This effectively lowers the execution overhead due to DFC. This number is expected to be smaller for the out-of-order (OO) IVM core due to superscalar execution there where some of the I/O latency can be hidden.

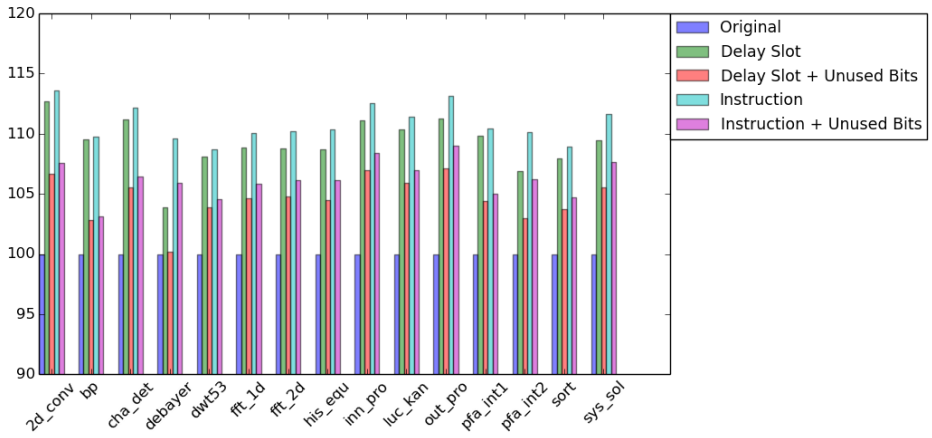

Figure 3.14: Leon core: Dynamic signature overhead for PERFECT benchmarks (normalized to original). Signatures can be embedded in unused delay slots (SPARC ISA cores) and unused instruction bits to lower overhead.

The detection latency of DFC is equal to the duration of the currently processed basic block. In the Leon core, the recovery latency, on the other hand, is determined by how fast the original content of the register file can be restored from its shadow copy. If a regular register file is used as a shadow copy, this would take more than 32 cycles. Instead, we propose to use a true monolithic shadow-register, with redundancy at the SRAM cell level that allows restoring the entire content in one cycle. In the IVM core, the recovery only involves restoring of the architectural state, normally accomplished in 1 cycle, while erroneous register and memory write requests are flushed from the re-order buffer (ROB) in a way similar to handling a branch misprediction.

3.7.3 Coverage

Static Branch Coverage

As described earlier, the DFC technique can only provide protection for basic blocks that are valid targets of direct branches, for which the signature can be generated. Therefore basic blocks originating from indirect branches, where targets are determined at run-time, as well as branches with targets outside of the binary (library and OS calls) cannot be protected. Furthermore, in our Leon implementation, few of the valid branches that did not have their delay slots preempted during the modified compilation process cannot be signed either. Table 3.2a shows a detailed breakdown of the types of branches and respective static compile-time counts (occurrences in the binary) for the Debayer PERFECT benchmark. We can see that most branches in a typical application binary are direct (85.7%). The 87.3% of direct branches have valid targets within the binary and 66.3% have preempted delay slots that allow signing.

Table 3.2: Leon core: Static branch instructions per branch type for Debayer PERFECT benchmark. Most branches are direct, have valid targets and preempted delay slots which allows protection of basic blocks.

Branch Type	All	[%]	Valid	[%]	Signed	[%]
Direct	3326	85.70	2905	87.34	2206	66.33
Bicc	2194	56.53	2191	99.86	1550	70.65
FBfcc	0	0.00	0	100.00	0	100.00
CVccc	9	0.23	0	0.00	0	0.00
Call	1123	28.94	714	63.58	656	56.63
Indirect	555	14.30	0	0.00	0	0.00
Jmpl	306	7.88	0	0.00	0	0.00
Rett	9	0.23	0	0.00	0	0.00
Ticc	240	6.18	0	0.00	0	0.00
TOTAL	3881	100.00	2905	74.85	2206	56.84

Static Executable Coverage

Table 3.3b shows a similar breakdown but in terms of code components that make up an executable. We see that while the source code (application and injection) is small in terms of the number of branches, most of them point to within their respective binaries and therefore can resolved for signing (89.4%, 93.4%). The library code (except for the injection library, which is mostly source code), on the other hand, has a much smaller number of such valid branches (30.5%), since most of them point outside the respective binaries, thus not allowing signing. The delay slot preemption that we added to the compiler works reasonably well. It frees 90.5% and 85.9% of delay slots for signatures in the user code (application and injection), respectively, and 67.3% in the Leon library. Finally, the executable that is made of the above binaries, receives signature coverage of 56.8%, which is between the above values, as expected.

Table 3.3: Leon core: Static branch instructions per executable component for Debayer PERFECT benchmark. Source code, as opposed to libraries, receives better protection due to larger number of valid branches.

Component	All	Indirect	Direct	[%]	Direct Valid	[%]	Direct Signed	[%]
Executable	3881	555	3326	85.70	2905	74.85	2206	56.84
Application Source	47	5	42	89.36	42	89.36	38	80.85
Injection Source	167	11	156	93.41	156	93.41	134	80.24
Leon Library	370	257	113	30.54	113	30.54	76	20.54
Injection Library	607	81	526	86.66	525	86.49	445	73.31

Dynamic Program Coverage

The actual coverage of application's execution time depends on the fraction of time spent in basic blocks that are signed and therefore protected. Figure 3.15 shows dynamic run-time counts (occurrences in execution) of basic blocks originating from different types of branches that determine their protection status. We can observe that, with an insignificant variation across benchmarks, most of the branches that execute are direct (87.7%, on average). Moreover, in spite of the lower, previously shown, static count, most of the executed branches are signed and protected (80.4%, on average). This illustrates that these are in fact mostly the protected branches from the source code components in the executable that dominate the execution and therefore result in coverage that spans most of the execution time. The remaining unsigned blocks are due to invalid targets or non-preempted delay slots in direct branches (7.3%, on average) and indirect branches (12.3%, on average).

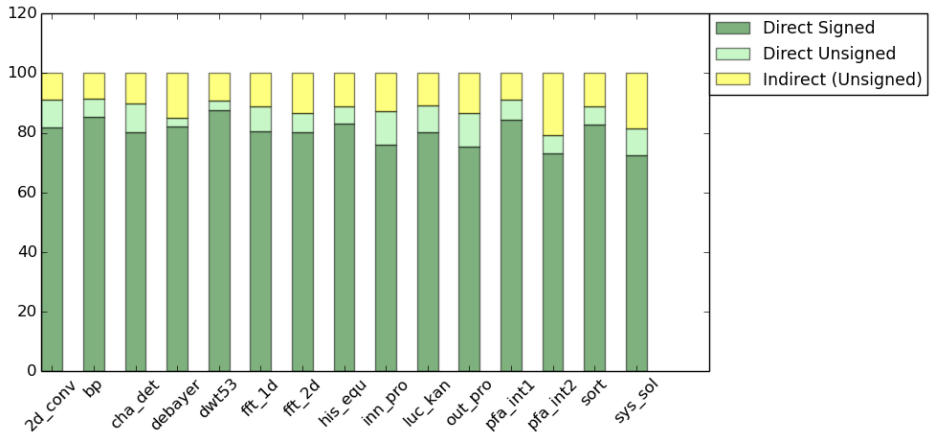

Figure 3.15: Leon core: Dynamic basic block count per branch type for PERFECT benchmarks (percentage). Most executed basic blocks are associated with valid direct branches which receive protection.

Error Type Coverage

The overall coverage capability of DFC can be illustrated by the overall detection rate for all tested benchmarks combined. For the Leon core, DFC detects 17.4% of all errors injected throughout the core. This confirms and outperforms results from the previous work [17] performed on a similar architecture. For the IVM core, our DFC model detects only 7.4% of all injected errors throughout the core. Previous work [31] of similar scope uses injection in the simulator, confined to particular structures, thus not allowing for comparisons. While most of the injected errors become meaningless (vanished and ONA, see methodology section) for both cores, there are more of those in the case of IVM (91.4%, Fig. 3.16, note the log scale) than Leon (79.3%, Fig. 3.17). The lower overall proportional detection rates are due to the fact that the significantly larger IVM core has many performance-oriented features that either do not affect the application's output or only occasionally distort the data flow detectable by the DFC checker.

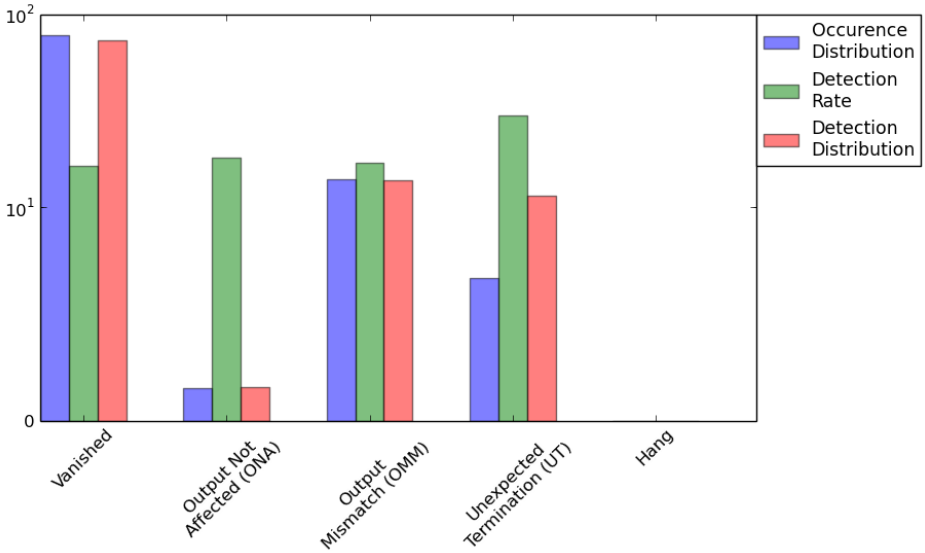

Figure 3.16: Leon core: Error distribution by type for PERFECT benchmarks (percentage, log scale). Most errors are vanished, detection rate is similar across all error types.

More detailed analysis shows that DFC in Leon provides similar detection rates for all error categories, with 21.2% for all meaningful errors (OMM, UT, Hang) and 16.4% for all meaningless errors (vanished, ONA), as shown in Fig. 3.16. DFC in IVM on the other hand, has a higher detection rate for the former (34.7%) than the latter (4.8%) as shown in Fig. 3.17. This is because much of the flip-flips naturally protected by DFC in the Leon core, especially in the back-end pipeline stages, are no longer meaningful to execution outcome while still receiving error detection. The checker in the IVM, on the other hand, verifies only the front-end of the pipeline, retire stage and related signals that are more essential to user-visible errors. The higher overall detection rate for meaningful errors in IVM is attributed to the fact that it detects more OMM

errors. Since these errors are usually related to computation, most of them are not visible to DFC. However, our results show that more of the vulnerable flip-flops in the IVM core affect both the computation and the control flow, thus increasing DFC detection.

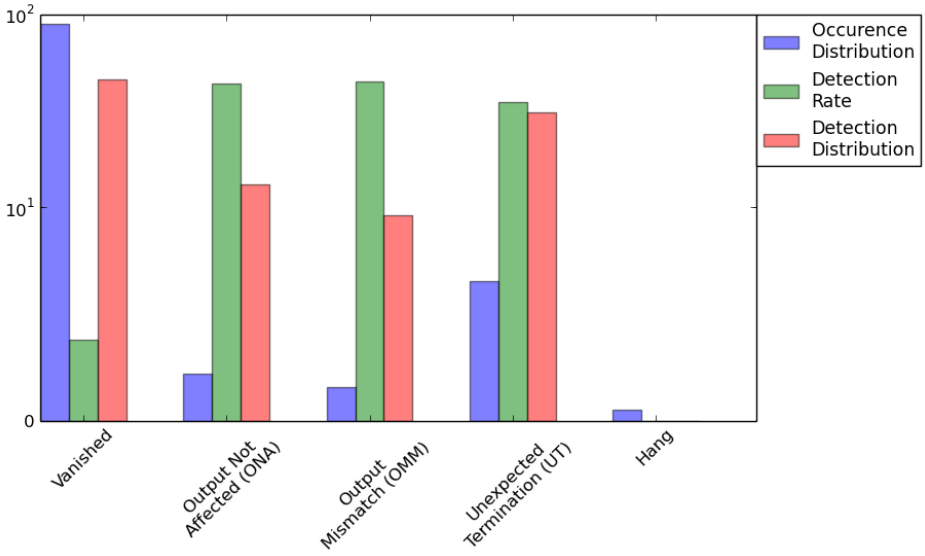

Figure 3.17: IVM core: Error distribution by type for PERFECT benchmarks (percentage, log scale). Most errors are vanished. Detection rate is higher for meaningful errors.

Component Coverage

Alternative protection techniques, such as hardening and parity, are applied to a particular circuit feature rather than activity in the program flow, like in the case of DFC. Therefore, for the purpose of evaluation and cross-layer design, it is important to determine what particular circuit features are protected by DFC. Moreover, it useful to consider only meaningful errors, which are those visible in the program output. This narrows the range of vulnerable flip-flops to only those that receive meaningful errors (82.7% in the Leon core). Within that range, 34.0% of flip-flops are at least partially protected (at least one error detected in each), with a corresponding detection rate of 53.2% (ALL category in Fig. 3.18). Analogously, for the IVM core, 64.2% flip-flops are vulnerable to meaningful errors only, and 75.1% of them are protected by DFC, with 39.7% corresponding average detection rate (ALL category in Fig. 3.19). However, with the overall protection range and detection rate, the overall vulnerability (soft-error-rate (SER)) improvement is higher for the IVM core (1.54x) than the Leon core (1.31x).

Since IVM is a much larger core with many performance-oriented features that either do not or only occasionally affect the application's output, as explained earlier, the core has a proportionally smaller number of flip flops vulnerable to meaningful errors. However, due to the nature of OO execution in the IVM core,

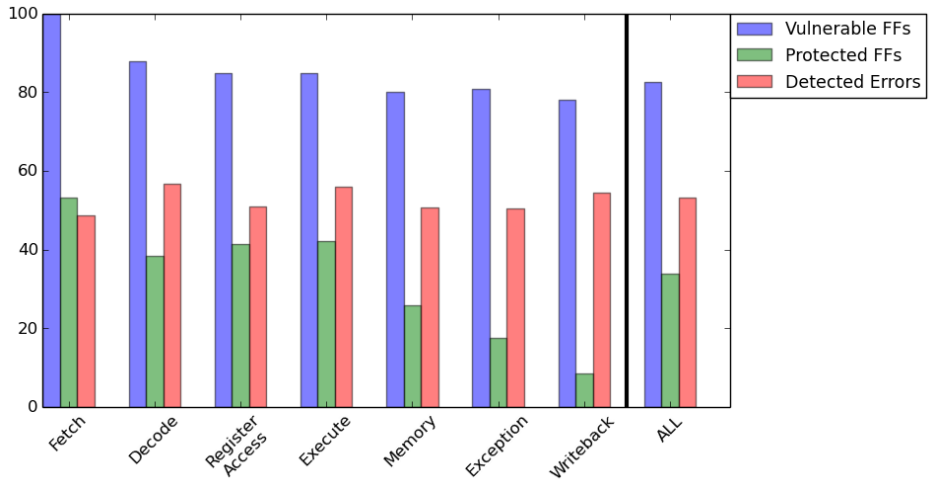

Figure 3.18: Leon core: Flip-flop protection per pipeline stage for PERFECT benchmarks (percentage). Front-end stages, used for data flow, receive wider flop-flop protection, while detection rate is similar.

there is proportionally a larger number of flip flops within that set that affect the control flow, both directly and indirectly, that are therefore protected by this technique. Since each of the flip-flops in this wider range is relevant to the data flow, and therefore detectable by DFC, even if only for a shorter period of time, the per flip-flop error detection rate is lower in IVM than that in Leon. Detailed analysis shows that the DFC checker placed in the writeback stage in Leon protects a similar amount of flip-flops across pipeline stages associated with forwarded instruction data used for checking 3.18). This is also illustrated by similar detection rates. In the IVM core, on the other hand, the DFC checker placed in the decode stage can only directly protect instruction data in the front-end stages as well as the retire stage, which is also closely involved in the flow determination 3.19).

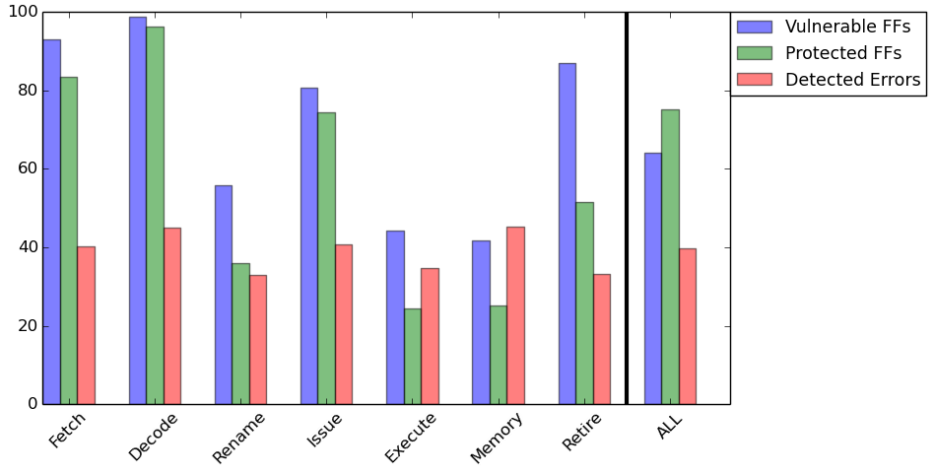

Figure 3.19: IVM core: Flip-flop protection per pipeline stage for PERFECT benchmarks (percentage). In OO IVM core, retire stage also receives wide flip-flop protection due to involvement in data flow.

Vulnerability Improvement

Fig. 3.20 and 3.21 show the flip-flop distribution in Leon and IVM designs in terms of vulnerability to meaningful errors for unprotected and DFC-protected designs (note the log scale). We can observe that, even in unprotected designs, most flip-flops have low vulnerability since the majority of errors are meaningless. Moreover, since the data flow and computation are more spread out across circuits and relevant cycles in the IVM core, there are fewer highly vulnerable flip-flops there. The addition of DFC removes many errors across the entire flip-flop range, which is also visible in the reduction of highly-vulnerable flip-flops. For the IVM core, DFC provides a smaller per flip-flop improvement but for a larger number of these features. The data shows that although partial protection of DFC might result in full coverage of some flip flops, most of them still suffer from many errors that are not detectable by this technique. Since most of the vulnerable flip-flops still remain in the system, additional or alternative protection may be required to reach desired SER targets. In fact, in the next section we show that alternative techniques provide much higher improvement at lower overhead.

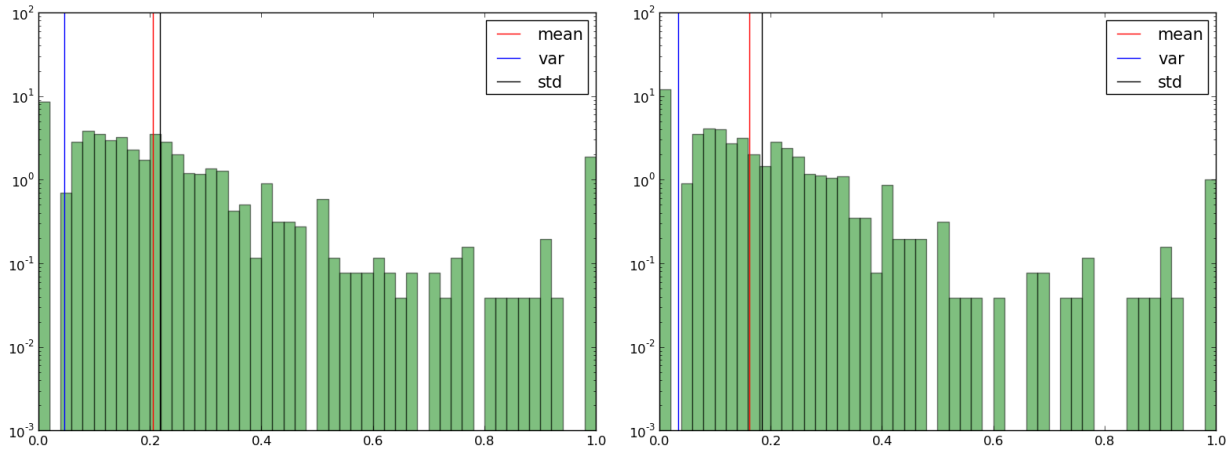

Figure 3.20: Leon core: Vulnerability of FFs a) without DFC and b) with DFC (percentage, log scale). In Leon, DFC protects relatively smaller range of flip-flops with higher vulnerability improvement in each, on average.

The two other, less capable, versions of DFC that we implemented include protection for branch instructions only (implemented in Leon) and protection for the PC only (implemented in IVM). These provide lower flip-flop protection range (13.2%, 31.4%) and similar detection rate (47.3%, 35.7%) for that range compared to the full DFC in these architectures. Therefore they are not competitive, considering similar area overhead incurred. For the IVM core, we considered an alternative checker placement in the Retire stage where only the PC and part of the instruction data (original register operand) are available after register renaming. While this placement has an opportunity to indirectly cover errors accumulated in previous stages, it does

not directly protect flip-flops in the front-end of the pipeline and other circuits involved in the data flow. The protection range and detection rate of this variant are 52.3% and 31.2%, which is worse than when the checker is placed in the Decode stage. Moreover, we combined the two checkers to determine that 94% of errors detected by the Retire stage checker form a subset of those detected by the Decode checker, making the former not worth the implementation.

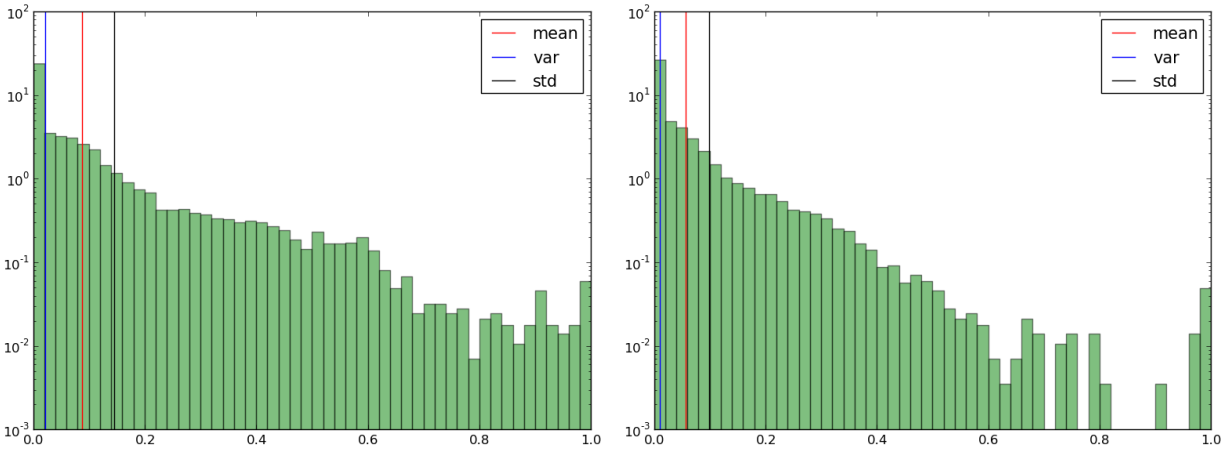

Figure 3.21: IVM core: Vulnerability of FFs a) without DFC and b) with DFC (percentage, log scale). In IVM, DFC protects relatively larger range of flip-flops with lower vulnerability improvement in each, on average.

3.8 Cross-layer Results

In this section we present characteristics of resilience solutions applied together in cross-layer combinations to Leon and IVM architectures. The analysis of solutions that involve DFC and ABFT are performed in collaboration with the Stanford University, while the remaining ones are provided entirely by our collaborators. In this final analysis, we only consider resiliency improvement for meaningful errors.

3.8.1 Hardware (Circuit-Logic)

The first cross layer solution that we evaluate involves a combination of circuit-level hardening and logic-level parity. These are generic techniques, applicable to any flip-flops. They can form a baseline for comparisons against the other, higher-level techniques, with more specialized coverage that we combine also combine with hardening and parity to form cross-layer solutions as discussed in the following sections. As mentioned earlier, we considered a variety of existing hardening techniques for flip flops that vary in terms of area, power and delay characteristics and can be characterized as light, moderate and heavy. To ensure better resilience

characteristics we mostly use one of the heavy variants that offers a significant reduction in the soft-error rate (SER) while roughly doubling the overhead of the protected flip-flip cell.

Similarly the Stanford group considers a variety of heuristics for the optimization of logic parity. Since the naive implementation can increase a design's critical path up to 10%, pipelining is used where needed, at the cost of increased detection latency, to avoid this effect. Although larger local flip-flop groups are preferred to reduce wiring overhead, the 16-bit group size is determined to be best in most cases, due to adverse delay effects for pipelining beyond that size. Due to the additional area and performance cost of pipelining, a grouping policy that takes into account multiple metrics, such as delay slack and groups size, lowers the overall overheads compared to optimizing for an individual metric.

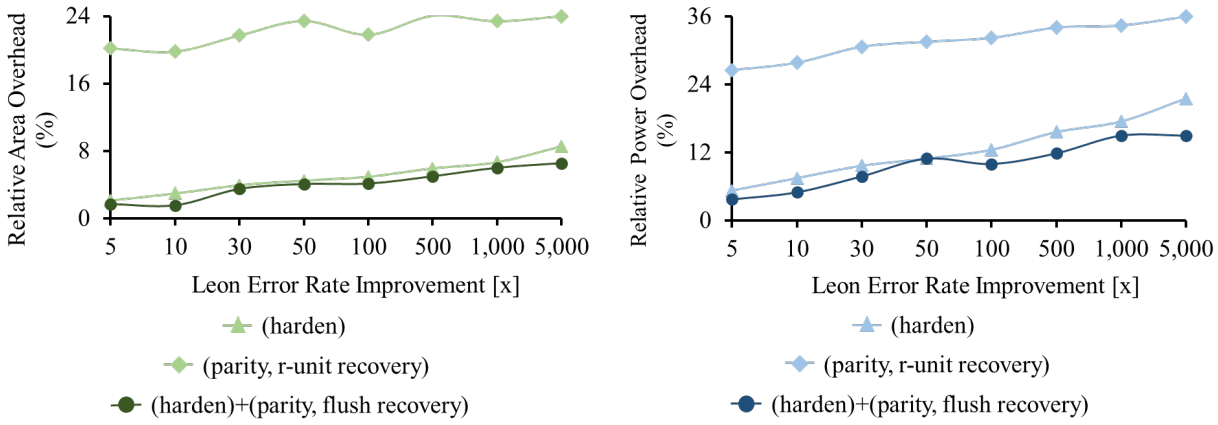

Figure 3.22: Leon core: overhead comparison of resiliency techniques in terms of a) area and b) power for a range of error rate improvement targets.

Since logic parity has a lower per-bit overhead for each protected cell, it is applied first as a part of this cross-layer solution at a potential slight loss of resilience (when considering multi-bit errors, for example). The technique is used for circuits where there is enough delay slack to insert the corresponding XOR tree. The remaining flip-flops, that still need to be protected to reach a given SER target, are hardened. The SER targets are achieved by protecting an appropriate amount of flip-flops, in the order determined by their vulnerability, that corresponds to the desired improvement. The recovery mechanism for parity considered in this solution involves the flush-based recovery, described earlier, that takes advantage of the pipeline flush capability in the core. This does not allow for using parity in last two pipeline stages (memory and retire), due to the long pipelined detection latency, where hardening is used instead.

We show that careful combination of hardening and parity protection for flip-flops, limited to most vulnerable flip-flops, allows designs that achieve significant savings as compared to a single-layer hardening-only or parity-only approach for all considered SER improvement targets (5-5,000x). From the Leon core (Fig. 3.22), we see that this cross-layer solution can yield anywhere from 1.1x-1.9x area improvement and

1.0x-1.5x power improvement as compared to the best single-layer solution (hardening-only approach in this case) while maintaining the original 2GHz design frequency. For the IVM core (Fig. 3.23), we see that a cross-layer solution can yield anywhere from 1.0x-1.4x area improvement and 1.0x-3.5x power improvement as compared to the best single-layer solution.

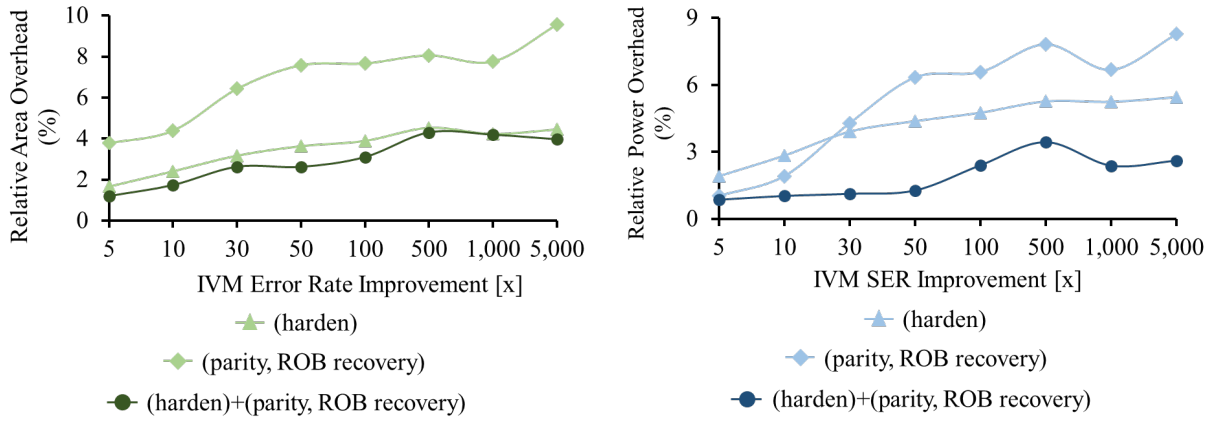

Figure 3.23: IVM core: overhead comparison of resiliency techniques in terms of a) area and b) power for a range of error rate improvement targets.

3.8.2 Hardware-Architecture

Having found a two-layer solution, we can use this as a baseline to evaluate the benefit of including another, higher-level, technique. Here we consider a three-layer hardware-architecture solution that includes Data-Flow Checking (DFC) on top of the previous two-layer solution of hardening and parity. The goal of this combined analysis is to determine whether DFC can provide more efficient protection for some flip flops. For this purpose, we add the DFC mechanism to both Leon and IVM architectures and determine the corresponding SER improvement and area overhead. We then determine the reduced vulnerability achieved with DFC for all flip-flops, compared to the two-layer solution, as well as the amount of additional protection with the two-layer solution needed, if any, to reach a range of SER improvement targets. The final overhead of the three-layer solution is determined by our collaborators based on the flip-flop vulnerability list that we provide. The list is generated based on error injection that we perform on the DFC.

As illustrated in the previous section, DFC alone provides an SER improvement of 1.31x and 1.54x for Leon and IVM cores, respectively, with no recovery considered. This improvement is below even the lowest, 5x, target that we consider with the two-layer solution, thus requiring significant augmentation to reach desired SER targets. However, the pipeline-level (no caches) area and power overheads of DFC (3.2%, 4.4%) in the Leon core are already higher than that of the two-layer solution (2.6%, 4.9%) even with no recovery. While the two-layer solution benefits from the recovery mechanism that is intrinsic (hardening) or included

in the core (parity), DFC in the Leon core requires a dedicated recovery unit. The addition of such unit increases the area and power overheads to 24.9% and 26.6% making this technique even more uncompetitive. Our data therefore show that the combination of DFC with the two-layer solution, for the protection of the remaining flip-flops, is always more expensive in terms of area and power for any SER improvement target (even when not considering expensive DFC recovery), as shown in Fig. 3.24. This is expected since, as explained above, DFC is less efficient in its protection domain than the two-layer solution.

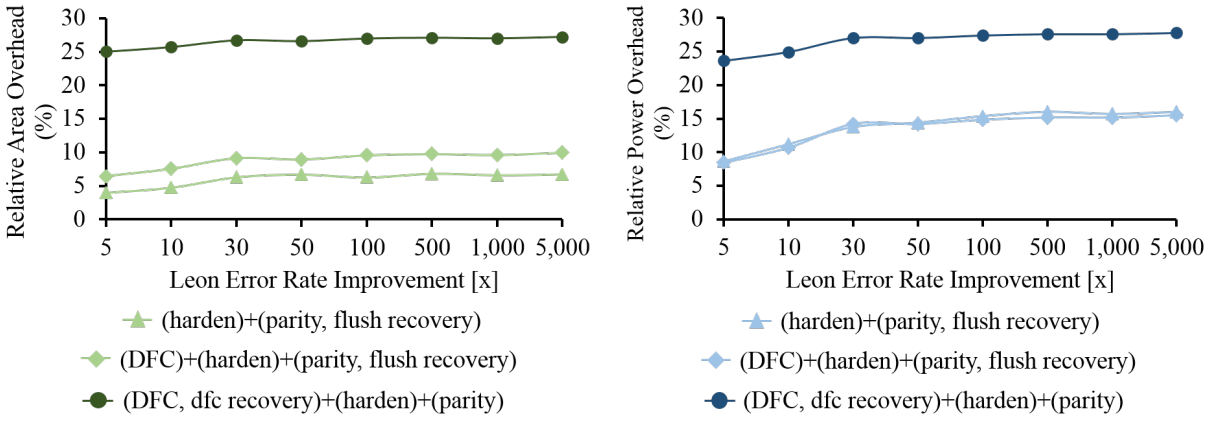

Figure 3.24: Leon core: Post-layout overhead comparisons: a) area, b) power

For the IVM core, the area and power overheads of DFC (0.15%, 0.19%) are lower than those of the two-layer solution (1.5%, 1.1%). The nearly negligible overhead of the checker and the recovery unit allows DFC in the IVM core to provide these marginal benefits. Further analysis shows a few points in the SER improvement spectrum where the three-layer combination outperforms the two-layer solution, mostly in terms of area (Fig. 3.25). However, our experiments indicate that these benefits lie within the range of noise resulting from minor differences in implementation and synthesis. Although DFC alone shows marginal area/power benefits over the two-layer solution, it provides a low SER that is easily achieved and outperformed by the two-layer solution. For the considered 5-5000x SER target range, DFC in Leon and IVM decrease the number of flip-flops requiring protection by only 9-11% and 2-7%, respectively. Finally, the performance overhead of at least some of the additional signature instructions makes this technique even more uncompetitive.

Although the three-layer solutions involving DFC are always worse in terms of coverage, performance, area and power, more careful examination of results discovers an interesting protection trend. Beginning with the lower improvement targets, such as 5x, achieving the corresponding SER reduction requires gradual protection starting from the most vulnerable flip flops. This is easily achieved with the two-layer solution that offers nearly full protection for each flip-flop at a relatively low overhead. Since DFC does not fully cover any of the flip-flops in its protection domain, it is not very applicable at the low SER targets. Achieving

the higher SER improvement targets, such as 50x and beyond, requires coverage of much larger groups of the remaining flip flops, including those that are less vulnerable, resulting in increased cost of the two-layer solution. This is the range of SER improvement targets where DFC helps. Since many of these remaining flip flops are within the scope of DFC, it can remove the few remaining errors there at a relatively lower overhead.

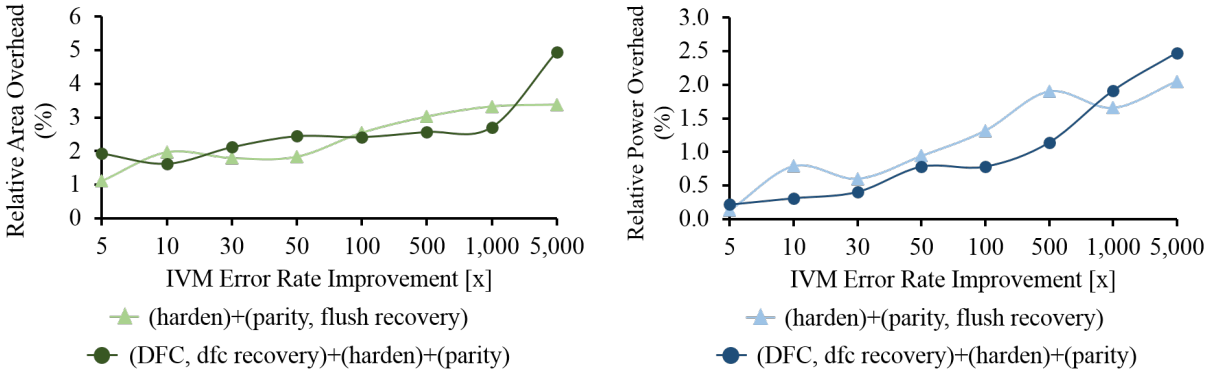

Figure 3.25: IVM core: Post-layout overhead comparisons: a) area, b) power

The essential reason why DFC proves not to be a competitive technique is the fact that it can only provides low, partial, protection for flip-flops that it covers. Unlike the alternative techniques that fully protect particular flip-flops, hardware units or entire execution threads, DFC protects activity in the program flow, which translates to intermittent protection of flip-flops. During cycles not protected by DFC, faults in these flip-flops can yield meaningful errors, hence requiring additional protection regardless of DFC. Our results show that the two-layer protection outperforms the DFC, proving to be a better solution alone. Moreover, this observation can be extended further. Even in most designs where protection is applied at a coarse-grained level of a unit, rather than a flip-flop, the minimum expected SER improvement for any component is always higher than what DFC can provide. Therefore the traditional approach of unit-level hardware replication or some form of instruction duplication, such as Error Detection with Duplicated Instructions (EDDI) or Redundant Multi-Threading (RMT), would be more viable solutions than DFC, even if they come at much higher overheads.

3.8.3 Hardware-Software

Similarly to the previous hardware-architecture three-layer solution, we subsequently consider a three-layer hardware-software approach that combines software techniques with the two-layer hardening-parity solution. For our analysis, we follow the same procedure as with the previous three-layer solution for the application of the considered techniques. Software-level techniques such as Algorithm-Specific Fault Tolerance (ABFT) are more effective at reducing hardware protection overhead since they target a notion of meaningful errors

in the context of a specific algorithm, such as FFT, for example. Therefore the goal of our analysis is to determine how many of the most vulnerable flip flops are sufficiently covered by this technique, and then how much of the additional two-layer hardening-parity protection is needed for the desired SER improvement range. Due to issues with the IVM compiler, we only present ABFT results for the Leon core. However, due to its dependence on the algorithm, we expect the efficiency of ABFT to be similar across architectures.

A mentioned before, for the purpose of completeness and to establish a better baseline for comparison, we analyze the previously proposed Control-Flow Checking by Software Signatures (CFCSS) and Error Detection by Duplicated Instructions (EDDI) techniques. For PERFECT and SPEC benchmarks we have looked at, as expected, our analysis does not show positive trade-offs for these solutions. It is due to their high performance overhead, which eliminates their inclusion in the cross layer solutions. Although these techniques do not incur area and power overhead from additional hardware, we see an application slowdown of on average 1.5x for CFCSS and 2.2x for EDDI. Furthermore, these software techniques tend to cover only a limited range of flip-flops (around 20%) which means that means that significant augmentation from circuit and logic-level techniques will be required to reach our SER improvement targets.

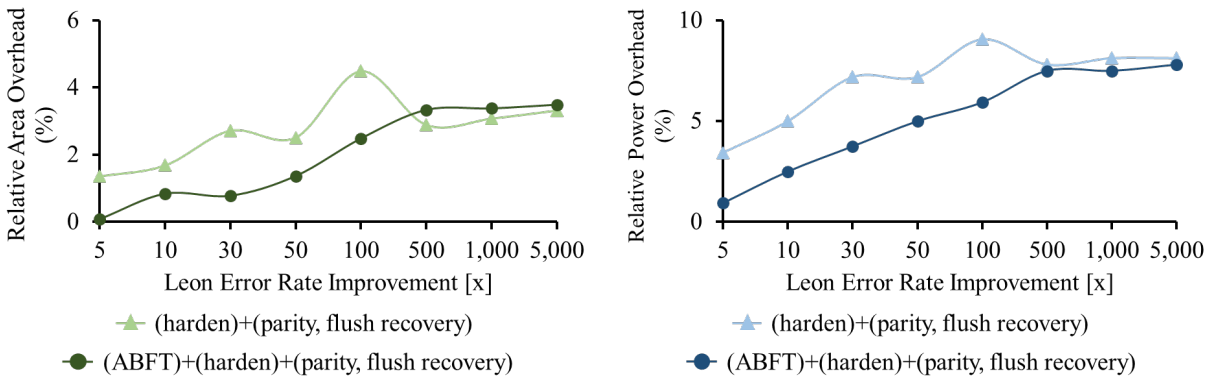

Figure 3.26: Leon core: Post-layout overhead comparisons: a) area, b) power

However, we determine that the Algorithm-based Fault Tolerance (ABFT) approach still remains beneficial to the cross-layer paradigm. Among the range of ABFT-protected benchmarks, we can distinguish those, such as matrix multiply, that provide both detection and correction on the fly via a checksum or a similar mechanism. We also have benchmarks, such as FFT, that only provide detection, thus requiring a recovery from a previously-taken checkpoint and/or recomputation of various amounts of data. The second group requires additional code to perform checkpoint-restart recovery and recomputation. This in turn changes the vulnerability characteristics of the application and the resulting overhead of two-layer hardening-parity protection used for the remaining unprotected flip-flops.

We see that a cross-layer combination of ABFT, hardening and parity, for benchmarks with correction

capability, actually achieves significant improvements as compared to the two-layer hardening and parity approach (Fig. 3.26). For resilience improvements of 5-100x, there is anywhere from 1.8-3.5x area and 1.4-3.7x power improvement. ABFT reduces the number of flip-flops that require hardware protection by 20-50%, thus reducing the overhead of the complementary two-layer solution. However, it gives negligible benefits for higher SER improvement targets (500-5,000x). While this technique provides high coverage for a small number of highly vulnerable flip-flops, the cost of comprehensive coverage of the remaining flip flop with the two-layer technique grows for the higher SER improvement targets. This effect is in fact opposite to what we saw with the three-layer solution involving DFC, where DFC provided better improvements at higher SER targets.

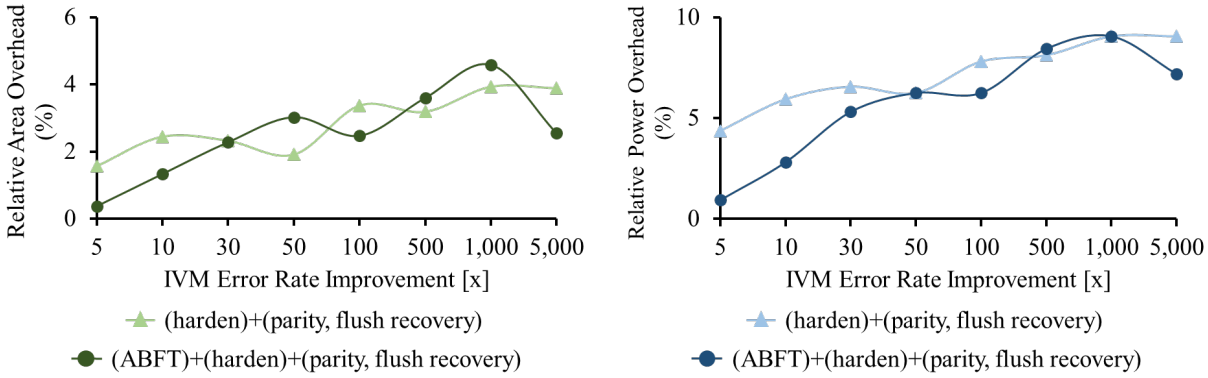

Figure 3.27: IVM core: Post-layout overhead comparisons: a) area, b) power

For the set of benchmarks with the detection capability only, we see that the three-layer solution involving ABFT still achieves significant improvements as compared to the two-layer hardening and parity approach (Fig. 3.27). For SER improvements of 5-30x, we see anywhere from 1.3-4.0x area improvement and 1.2-4.7x power improvement. However, we observe that for the specific set of applications that we evaluate here, the corresponding ABFT algorithm is weaker, meaning that it provides less detection for the touched flip flops. Moreover, since this version of ABFT merely detects errors, it may be inappropriate for many scenarios where long-latency recovery is unacceptable (i.e., in embedded, real-time systems). However, for low resilience improvement targets, in which longer recovery latency is tolerable, ABFT reduces the number of flip-flops that require hardware protection by 15-50%.

3.8.4 Hardware-Architecture-Software

Finally, we analyze the four-layer solution that involves DFC and ABFT combined with the two-layer solution of parity. As illustrated above, DFC alone is more efficient at higher SER improvement targets, while ABFT performs better at lower targets. Moreover, DFC provides better protection against UT (unexpected

termination) and hang error outcomes, common results of control-flow failures, while ABFT has a higher coverage of OMM (output mis-match) outcomes. We want to explore the combination of these two techniques together with the two-layer hardware solution, to determine if they can augment one another, resulting in higher coverage of meaningful errors while reducing further protection. We assume no overhead for the DFC recovery unit to make it more representative of the DFC overhead in the IVM core. We already determined that inclusion of the recovery unit in Leon would always make DFC cost-ineffective. Similarly to the previous section, the results here are given only for the Leon core. However,

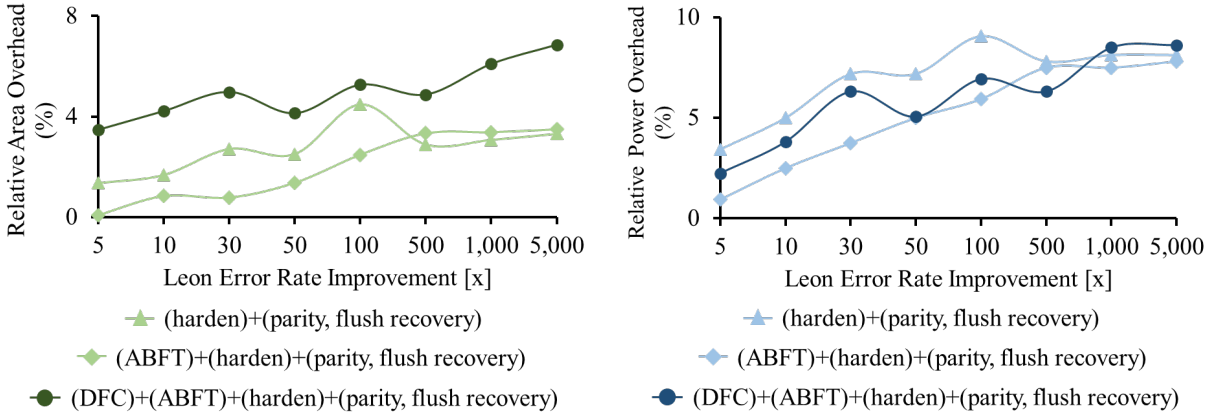

Figure 3.28: Leon core: Post-layout overhead comparisons: a) area, b) power

Similarly, due to ABFT considerations, we first evaluate this cross-layer combination with a set of ABFT benchmarks that have correction capability. We discover that the addition of DFC to the three-layer combination of ABFT, hardening, and parity yields essentially no benefit, while in most cases it is even worse than this three-layer combination. The DFC complements the three-layer solution by decreasing the number of flip flops requiring two-layer hardening-parity protection by an additional 10-20%. However, the area and power costs of the DFC checker itself negates these and is in excess of these additional benefits that it provides over the three-layer solution (Fig 3.28). Moreover, while this assumes the negligible cost of recovery, such as that in the IVM core, the Leon implementation would incur significant associated overheads.

Although we already see that cross-layer combinations of DFC with ABFT-correction and low-level hardware yield no benefits, for completeness, we also consider the same combinations with the set of ABFT applications that provide detection only. Similarly, we discover essentially no benefit of adding DFC on top of the three-layer ABFT-hardening-parity solution and in many cases even a degraded performance. DFC does reduce number of flip-flops requiring low-level hardware augmentation by an additional 3-15%, but this additional reduction is insufficient to justify the DFC hardware cost incurred by adding this fourth-layer solution (Fig 3.29).

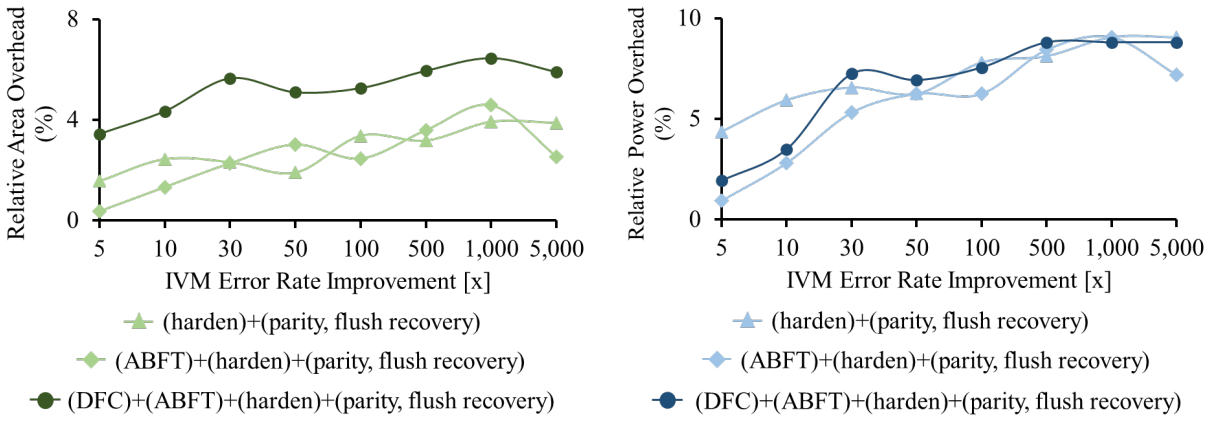

Figure 3.29: IVM core: Post-layout overhead comparisons: a) area, b) power

3.9 Benchmark Dependence

In this section we investigate additional benchmark considerations to better understand the effectiveness of the analyzed protection solutions.

3.9.1 Resilience Variation

In previous sections we determine flip-flop vulnerability based on a combination of all PERFECT (DFC, ABFT, hardening, parity) as well as SPECINT benchmarks (hardening, parity). The single or cross-layer resilience is then added to flip-flops in order determined by the obtained vulnerability hierarchy. This is a good design practice, as it results in a protection of features utilized by a variety of benchmarks. However, we can think of cases where a diverse benchmark suite is not available, or where it is desired to protect the design only against vulnerability specific to certain workloads. Therefore we wanted to illustrated the difference in effective resilience that might be observed when different sets of benchmarks are used at design and evaluation times. To that end, we devised a series of experiments to analyze the role of benchmark choice as well as their dependence. Since it is hardening and parity that are mostly utilized in the cross-layer combinations, for reaching soft error rate (SER) targets, we do this analysis for this two-level technique alone. We use the Leon core and SPECINT benchmarks.

3.9.2 Training/Evaluation Sets

We begin by exploring the dependence of benchmarks with our cross-layer approach by utilizing training and evaluation sets commonly used in the community. Using our 11 SPECINT benchmarks, we create training sets by randomly selecting 4 of the 11 benchmarks for use and reserving the remaining 7 benchmarks for the evaluation set. We then do the vulnerability analysis and add resilience based on the hierarchy reported by

our training set. Finally, we evaluate the resulting resilience by running benchmarks in the evaluation set instead. From this data, we can determine the difference between how much resilience we had targeted (and had expected to achieve) compared to the actual resilience we achieved in practice on a different workload.

To bridge the gap between the targeted and achieved resilience, we can evaluate using two scenarios: match and augment. For the match scenario, we continue adding protection to flip-flops, based on information from the training set, until the resilience difference is matched and the design SER target is reached. For the augment scenario, we lightly-harden (around 50% area overhead per flip-flop) all other flip-flops not protected based on information from the training set. Table 3.4 depicts the average for the 10 random training/evaluation set pairs we analyzed for various resilience targets as well as the corresponding match/augment scenario results.

From these results, we see that for low resilience targets, there is only a minor underestimation of resilience improvement. However, for mid to highly resilient designs, training and evaluating using different applications can lead to drastically underprotected designs. However, we also observe that by simply augmenting our design (lightly-hardening all unprotected flip-flops), we can guarantee that our design will meet or surpass the intended resilience target aside from the most highly-resilient design.

Table 3.4: Leon core: Resilience improvement applied according to results from the training SPECINT benchmark set, evaluated on another set and matched/augmented to reach the original design SER targets.

Training	Evaluation	Match	Augment
5.0x	4.8x	5.0x	19.0x
10.1x	9.3x	10.0x	36.9x
50.2x	36.6x	50.3x	143.5x
100.6x	61.0x	101.3x	236.0x
506.4x	324.4x	515.0x	1,064.4x
1,008.1x	513.4x	1,010.0x	1,499.9x
5,064.1x	1,226.8x	5,406.3x	2,492.4x

3.9.3 Benchmark Similarity

To better understand why results obtained from training and evaluation set experiments, we also analyzed benchmarks to determine commonalities. As mentioned earlier, for every benchmark, we create an ordered list of flip-flops based on vulnerability (using the metrics defined in methodology section). This allows us to determine whether or not the relative importance (i.e., vulnerability) of particular flip-flops remain consistent across benchmarks. Using the SPECINT benchmarks, we can divide each ordered list into deciles, for example, and determine how many of the flip-flops in each 10% section are shared among all benchmarks.

From Table 3.5, we see that the most (0-10% decile) and the least (90-100% decile) vulnerable flip-flops are fairly similar and consistent across all benchmarks. This means that the most and the least vulnerable flip-flops can be attributed more to the specifics of an architecture and its functionality, rather than to specific applications. This similarity, across all benchmarks for the most vulnerable flip-flops, explains why we are able to closely match a low resilience target without any additional protection, despite the fact that our training and evaluation sets differ. Similarly, improving the error rate for the highest SER targets, which involves the coverage of flip-flops within 90-100% decile, does not incur much overhead due to commonalities across benchmarks. However, we see that because the moderately vulnerable flip-flops do not share commonalities across benchmarks, evaluating resilience using benchmarks other than the training set will yield significantly underprotected designs for higher resilience targets.

Table 3.5: Leon core: Shared flip-flops across SPECINT benchmarks according to vulnerability (most to least vulnerable). Most and least vulnerable ranges are common across benchmarks.

Vulnerable Decile	FFs Shared by Benchmarks		
	by all	by all, but 1	by all, but 2
0-10%	83.33%	88.10%	94.05%
10-20%	4.76%	17.86%	36.90%
20-30%	0.00%	1.19%	4.76%
30-40%	0.00%	5.95%	15.48%
40-50%	0.00%	3.57%	17.86%
50-60%	0.00%	0.00%	1.19%
60-70%	0.00%	0.00%	1.19%
70-80%	0.00%	0.00%	3.57%
80-90%	0.00%	8.33%	28.57%
90-100%	11.90%	46.43%	70.24%

3.9.4 Expected Resilience

Based on what we have shown, we see that with the design based on a training set, it is possible to provide resilience for all other evaluation sets using the a) match and b) augment scenarios. With these, we can either a) use margins to more or less match design targets for all evaluation sets or b) lightly harden the remaining unprotected flip-flops. While matching is more accurate when targeting an particular evaluation set, it is difficult to strictly bound the number of extra flip-flops that need protection when considering all other evaluation sets. We have found that we typically need to add protection to anywhere between 3-18% additional flip-flops (Fig. 3.6) to provide expected level of resilience for all evaluation sets. Due to similarities between vulnerable flip-flops across benchmarks (Table 3.5), the incremental overhead is small for low and larger for mid-range SER targets.

The uniform application of light-hardening to all unprotected flip-flops, on the other hand, is straightforward but imprecise since it does not take into account vulnerability ranking of flip-flop in the training or evaluation sets. This approach applies, less robust, light hardening across all remaining flip-flops with various vulnerabilities. As a result, a significant number of flip-flops needs to be hardened to decrease the SER rate enough to reach even the lowest SER targets. Compared to matching, this causes an unnecessary area and power overhead at lower SER targets. For higher SER targets, the most vulnerable flip-flops are already covered in the original design, while light-hardening can cover the remaining, less vulnerable, flip-flops at a low overhead. This scenario is therefore beneficial for mid and high resilience designs targets.

Table 3.6: Leon core: Relative area and power average overheads for expected resilience. Match and augment method are more efficient at low and high resilience targets, respectively.

Target	Match		Augment	
	Area	Power	Area	Power
5x	0.31%	3.66%	63.74%	32.32%
10x	1.53%	3.77%	44.56%	22.64%
50x	7.21%	7.03%	20.10%	11.50%
100x	6.43%	6.19%	16.96%	9.44%
500x	12.07%	12.59%	9.58%	5.54%
1,000x	14.44%	15.19%	6.89%	4.44%
5,000x	20.91%	16.60%	4.19%	2.10%

3.10 Conclusions and Future Work

In this chapter, we performed a detailed analysis of the Data-flow Checking (DFC) architecture-level resilience technique in cross-layer combinations with hardware-level hardening and parity as well as software Algorithm-Based Fault Tolerance (ABFT). The first goal of our work was to determine implementation complexities and the applicability of DFC. The DFC design involves many choices with respect to the placement of the checker, availability of data-flow signals and architectural specifics, all of which result in varied coverage. The additional challenges include compiler modifications to analyze the the data-flow graph for signature generation and to embed signature instructions in the binary. Unlike hardening and parity, but similarly to ABFT, this approach partially covers only a specific range of flip flops (related to data flow in this case) and basic blocks (originating from direct-branches only). The lack of full coverage and performance overhead (at least some additional signature instructions) disqualify this approach for being used in the industry that favors straightforward designs with predictable coverage, even at a higher overhead.

Another goal of our work was to analyze the area and power efficiency of DFC alone, as well as determine its benefits for cross-layer resilience combinations. Together with our collaborators from Stanford University,

we show that determination of vulnerability at the flip-flop level allows for selective application of low-level techniques such as hardening and parity at much lower overheads than those of traditional coarse-grained solutions. We illustrate that, for the area overhead of DFC, the soft error rate (SER) improvement provided by this solution is about an order of magnitude lower than that achievable with hardening and parity (and even worse for small cores due to DFC recovery overhead). While DFC and ABFT are more effective at different ranges in the SER spectrum, only ABFT proves to be efficient and yields meaningful benefits in combinations with the low-level techniques. Since most designs would expect SER improvements higher than that of DFC, we conclude that alternative protection would always be required thus obviating the use of DFC.

Our results suggest two directions for improving resilience. More reliable devices, such as hardened latches, employed selectively give better protection than architecture-level, more complex, techniques added to unreliable components [54]. Only specialized protection that specifically targets vulnerable features, such as the ABFT approach, proves to be optimal for applicable designs. Since coverage of architecture and software-level techniques is confined to a specific range of flip-flops and specific cycles, they can protect against multi-bit errors (errors in adjacent cells) only within those constraints. While we can achieve multi-bit error coverage for two adjacent hardened cells, this cannot be obtained with parity. Although particles such as muons differ in nature from typical radiation [55], they do not significantly change protection design trade-offs. Since comprehensive fault injection for large designs, common in industry, is time consuming, hierarchical approaches that rely on statistical vulnerability/propagation properties of components [56] may need to be used in those cases. Future research should focus on designing more effective architecture-level solutions and enabling optional hardware protection on demand. Moreover, the cross-layer paradigm could benefit from a larger number of protection techniques as well as from the reconciliation of high and low-level fault injection methodologies.

Chapter 4

Cross-Layer Protection in an Accelerator

4.1 Overview

With the current trend of maximizing performance within a fixed power budget, future generations of processors are expected to reserve more silicon area for accelerators and configurable logic to optimize some of the common operations. The complexity of accelerators varies depending on the scope, which includes individual instruction types (co-processors, arithmetic units), entire algorithms (ASICs) or generic computation (GPUs). The architecture of these accelerators also differs in the number of computation units and latency, depending on performance expectations. Some of the same generic resiliency techniques, such as hardening, parity and replication, which we studied in Chapter 2 and 3, apply to accelerators. However, since these units perform specific functions, they could potentially benefit more from the corresponding hardware implementations of Algorithm-Based Fault Tolerance (ABFT) protection techniques, if available. Furthermore, since accelerators are smaller in size, simpler in structure and don't involve the user code at the software level, they exhibit different vulnerability and protection overhead for the same functionality, when compared to general-purpose cores. Therefore, analysis similar to that in Chapter 2 and 3 should be also performed when designing optimal protection for an accelerator. It would be useful in illustrating differences in trade-offs between familiar protection techniques in the context of such a special-purpose unit.

In Chapter 4, we address the above research questions by analyzing a subset of the previously studied protection techniques, including hardening, parity, replication and ABFT in particular [10], for the FFT accelerator. We select FFT as an example accelerator because it is one of the most popular and widely

available units. It has a complex structure with a significant number of producing and consuming units that exchange data, which in turn increases the vulnerability of the output due to potential error propagation. Moreover, FFT has a simple and a more sophisticated versions of ABFT protection algorithm that can represent two extremes in the overhead spectrum. We first extensively modify and modularize FFT core, for the application and analysis of the aforementioned resilience techniques. We then implement these solutions in RTL and add them to the corresponding units in the accelerator. Finally, we develop a fault injection infrastructure and a retry recovery mechanism for the considered techniques. The overall area overhead is analyzed for multiple accelerator and ABFT configurations that involve varying computation capability of processing units. The core vulnerability and protection effectiveness are determined via fault injection, similar to that in Chapter 3. Errors are analyzed with a breakdown per type and affected accelerator component.

In spite of the extensive previous work on various special-purpose processing units, as well as their increasing popularity in the form of an accelerator, we have not see any case studies comparing various protection techniques, including ABFT, for an accelerator. The first goal of our work is to determine the vulnerability of the FFT computation in the accelerator. This allows us to illustrate architectural differences in flip-flop vulnerability since, unlike in general-purpose cores, most accelerator resources are highly utilized for computation. The analysis also illustrates vulnerability of the algorithm in the accelerator, since most of the flip-flops in an accelerator are used in computing the output. The second goal of our work is to show the effectiveness of resilience techniques, especially hardening/parity and ABFT. We expect the efficiency to be higher in the accelerator due higher vulnerability confined to smaller number of flip-flops. Our results illustrate that, in contrast to general-purpose cores, the majority of the flip-flops are relevant to FFT computation and meaningful errors visible in the output. We show that due to the high overhead of full ABFT, only its simple version combined with hardening/parity provides the most optimal protection. Our analysis also explores different area optimizations for the FFT core and the checker. This chapter makes the following contributions.

1. We implement a variety of resilience solutions for the FFT accelerator including hardening/parity, replication, and ABFT.

2. We modify the FFT core for better application of resilience techniques and integration with the retry recovery solution.

3. We analyze area and performance overhead of these techniques for multiple configurations of accelerator and protection (ABFT).

4. We develop a fault injection framework to evaluate accelerator vulnerability and the efficiency of the considered protection.

5. We discuss differences in logic vulnerability between our accelerator and general-purpose cores used in Chapters 2 and 3.

4.2 Previous Work

In this section, we describe previous study of resilience in hardware accelerators. The concept of using a special-purpose unit to accelerate functionality of the main core can be traced back to coprocessors such as those for I/O or arithmetic operations in early CPUs [57]. Many of the synthesizable cores include an optional hardware floating-point unit, which poses a design trade off due to its significant size [47]. The lack of such hardware computation capability will require instructions to be processed in software at a much lower speed. Today, accelerators are typically used for optimizing performance of entire algorithms, such as DSP, sorting or image processing, rather than individual arithmetic operations [21]. There is a variety of special-purpose units that are used in media or imaging equipment alone or integrated as a coprocessor with the CPU in devices such as cell phones [58]. Graphics Processing Unit (GPU) is an example of a large-scale accelerator that can also execute general-purpose workload, similarly to CPU, but at a higher efficiency. Moreover, the current trend suggests including reconfigurable logic, Field Configurable Logic Arrays (FPGAs), on the processor die for the acceleration of desired functionality. We have recently observed collaborations between the mainstream CPU and FPGA companies to become more proficient in this area. Moreover, there is a body of work dedicated to distributing work between CPU and accelerators to save power under the dark-silicon concept [59].

In spite of the significant existing body of work on the topic, we are not aware of any previous attempt to compare various protection techniques for any of the existing accelerators. Similarly to general-purpose cores, most previous work considers traditional protection approaches with parity or replication for the protection of logic as well as parity or SRAM for the coverage of memory arrays. In addition to those, in our work we consider hardening, proposed in [14], as well as logic parity, presented in [20]. The Algorithm-Based Fault Tolerance (ABFT) that we also analyse, was first proposed for the matrix multiplication [18] and then extended to the Fast Fourier Transform (FFT) [10] and other operations with verifiable computation stages. All of these have only software-based ABFT implementations for the protection of the corresponding algorithms. However, software execution in a general-purpose processor involves a larger number of flip-flops as well as it additionally suffers from errors in the handling of instructions and execution of the OS/library code. Therefore these implementations have few similarities to special-purpose units in terms of potential

benefits of ABFT. The only existing hardware resiliency implementations for FFT [45] rely on redundancy for both detection and recovery. This solution reexecutes the workload to detect soft-errors while utilizing an alternative execution path for the detection of hard-errors.

4.3 Platforms

In this section we give an overview of the computation involved in the FFT algorithm and its breakdown into units of work that can be mapped to processing units of various complexity. We describe hardware implementation options that differ in terms of resource requirements and performance. Finally we give details about the FFT core that we used for the fault tolerance evaluation.

4.3.1 Fast Fourier Transform (FFT)

The Fast Fourier Transform (FFT) algorithm is an efficient implementation of the Discrete Fourier Transform (DFT) that approximates a time-domain function with a series of sinusoidal components of different frequencies and amplitudes for more convenient processing. FFT significantly reduces the algorithmic complexity of DFT by taking advantage of symmetry in computation. The computation of each FFT output is based on all of the inputs that are added and multiplied by weight factors in a particular fashion as illustrated by the butterfly diagram (Fig. 4.1). The calculation can be broken down to smaller butterflies of different granularities, such as 2x2, 4x4, 8x8, etc., where each involves the same type of computation but on a different set of inputs and weights. The number of stages in the diagram is given by Eq. 4.1.

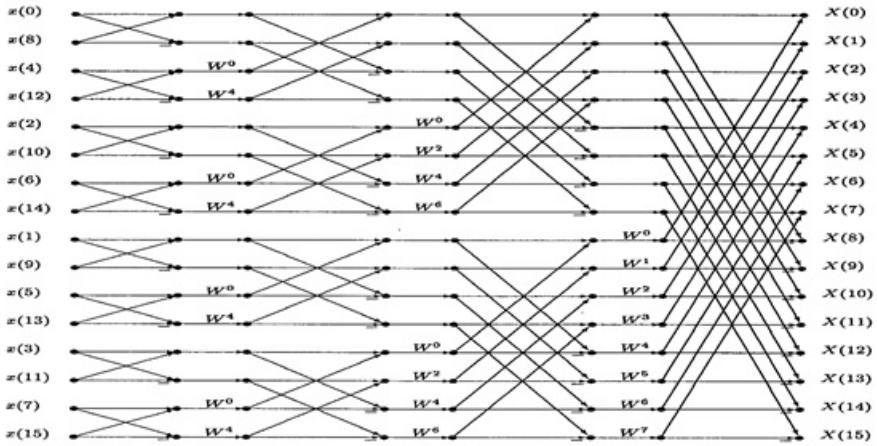

Figure 4.1: FFT "butterfly" computation diagram for a 16-bit input sequence. Each unit work involves the same computation on different data [9].

$$\#\ \text{of stages} = \log_2 \mathrm{N} \tag{4.1}$$

4.3.2 Theoretical Full-Size Implementation

The theoretical full-size hardware implementation of FFT would resemble that in Fig. 4.2. There would be a separate hardware unit dedicated to each unit computation in the butterfly, resulting in a significant overall number of computation resources given by Eq. 4.2, where N is the size of the input/output data sequence. Such a system could either be pipelined to process FFT stages in parallel, or process data in a single cycle (no registers required to store intermediate data). If no continuous data is supplied, many of the processing units would be underutilized. Although the speed of processing is a benefit of a full-size implementation, a simple unit clocked at a higher frequency could be used instead to reduce size. If acting as an accelerator that reads from memory, the speed of processing has to be adjusted to memory access time, which in turn removes any performance benefits of a full-size implementation. Finally, due to a large number of arithmetic units across all FFT processing elements, the area overhead of full-size implementation could exceed many-fold even that of a general-purpose core.

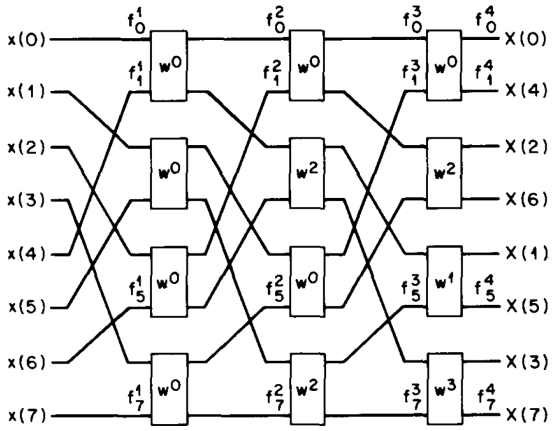

Figure 4.2: Theoretical full implementation of an 8-bit FFT butterfly with dedicated processing units for each unit computation [10].

$$\#\ \text{of units} = \text{rows} * \text{columns} = \frac{\mathrm{N}}{2} * \log_2 \mathrm{N} \tag{4.2}$$

4.3.3 Practical Single-Unit Implementation

The more practical hardware implementations of FFT used in many real-world application consist of at most a couple of cores with various computation capability. We have determined that, among those, the most

popular cores have the half-2x2, full-2x2 and two-2x2 FFT unit structure, where some pipelining might be used for independent computations [58]. We discuss the relative area in the results section. The core used for this work, obtained from [21], features the half-2x2 FFT unit structure. It includes a single processing unit with half the capability of the full-2x2 core (Fig. 4.3), that takes two cycles to process the unit work seen in Fig. 4.2. We selected this particular core due to its straightforward implementation that is applicable to our asynchronous accelerator model (explained later). Moreover, the core came with an algorithm-equivalent software implementation that we used for verification purposes. The architecture is capable of processing a 1024-bit complex (i.e., real plus imaginary) integer sequences. Due to the significant size of the floating-point units, most FFT accelerators work with integer arithmetic. Floating-point implementations are feasible mostly in software that runs in a general-purpose core where such unit is usually present.

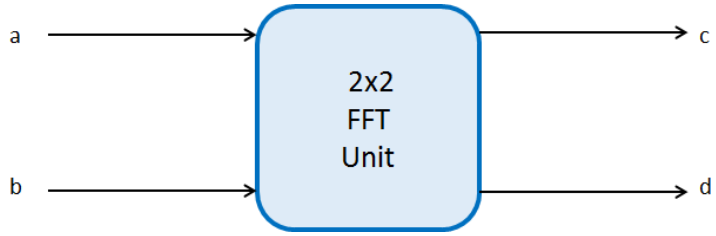

Figure 4.3: More practical implementation of an FFT butterfly with a single 2x2 processing unit for recursive processing of data.

$$c = a + b * W_N^k \tag{4.3}$$

$$d = a - b * W_N^k \tag{4.4}$$

The operation performed by a full-2x2 FFT core involves two complex additions and one complex multiplication according to Eq. 4.3 and 4.4 as well as Fig. 4.4, where W is the appropriate weight factor. Each complex addition translates to two real additions (Eq. 4.5). Each complex multiplication translates to four real multiplications and two real additions (Eq. 4.6). In order to perform these operations in one cycle, each requires a dedicated hardware unit. However, our core optimizes area at the cost of performance by employing only 2 multiply units shared across the 2 computation cycles that are now required to process unit work. Depending on performance expectations, this optimization might be tolerable. Assuming similar frequency, this computation is at least an order of magnitude faster than in a general-purpose core, where each operation would need to be performed as a separate instruction.

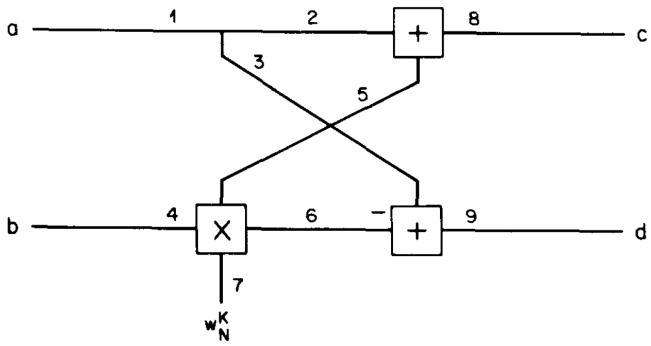

Figure 4.4: Multiplication and addition operations involved in a 2x2 unit computation on a pair of inputs in the FFT butterfly [10].

$$(a + ib) + (c + id) = (a + c) + i(b + d) \tag{4.5}$$

$$(a + ib) * (c + id) = (a * c - b * d) + i(a * d + b * c) \tag{4.6}$$

The FFT core that we use assumes asynchronous processing, where the task is issued by the main processor that previously stores input data in memory. In this version, there is only a 1-cycle initial and final delay associated with reading the first and writing the last element from/to memory, respectively. In the case of real-time processing, not considered here, there is an initial delay while the FFT unit is waiting for the first pair of data to become ready for processing, equal to N/2, where N is the number of inputs. The waiting time for the remaining data is overlapped with computation. There is also a final delay due to sequencing of output data that is normally not computed in order. In both cases, given a typical small number of processing units (half to two), the computation by far dominates the execution time.

4.3.4 Structure of the FFT Core

The complete core includes the address generator, the aforementioned FFT butterfly unit, RAM controller, trigonometric ROM array as well as input and output RAM arrays (Fig. 4.5a). Since the core includes a single processing unit, an address generator is used to create addresses for input, trigonometric and output data that is appropriate for the section of the butterfly being processed. The FFT unit performs the basic butterfly computation as described above. The RAM controller is used for switching access to data arrays between the external processor and the FFT core as well as switching input/output ports during FFT computation. The trigonometric ROM, with a single read port, contains a sampled sine/cosine data used for the generation of

weights in the butterfly (and the ABFT checker). Input and output RAMs, with 1 read/write port each, are used for storing complex input/output data. We can observe that 1024-wide data arrays take 58% of the core's area. This fraction would change if more compute units were employed in the accelerator. Due to relatively large multiplier circuits, the FFT unit is significantly larger than the remaining, address generator and RAM controller, logic (80% of all logic circuits). However, the remaining logic has a proportionally larger amount of flip-flops, which decreases the relative contribution of the FFT unit in terms of flip-flops to 68.9% (Fig. 4.5b). The relative power consumption in the accelerator and its protection circuitry is very correlated with area due to high utilization of these units (not always the case with general-purpose core). Therefore we do not include additional data and plots for power in this chapter.

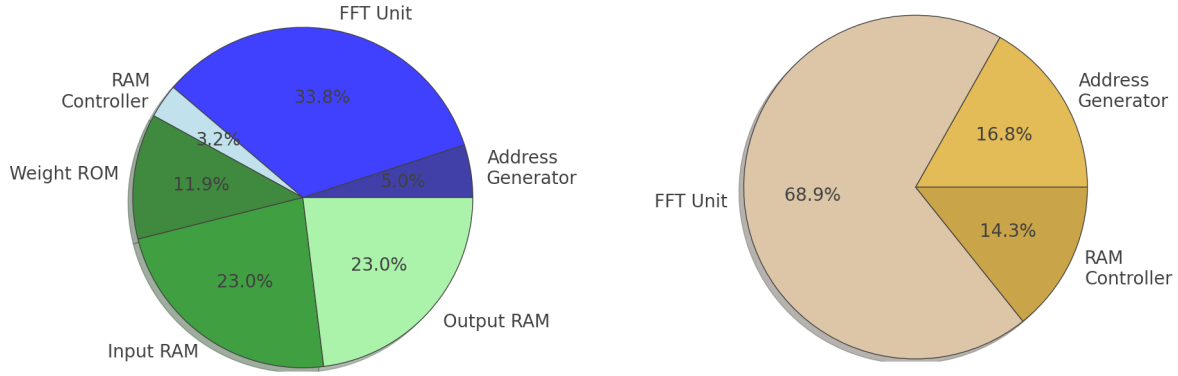

Figure 4.5: Contribution of components in a half 2x2 FFT core in terms of a) area and b) flip-flips (logic only). SRAM arrays dominate the area.

4.4 Analyzed Protection Techniques

In this section we give an overview of the protection techniques considered for the FFT core, which include hardening, parity, replication and ABFT. We also describe major implementation considerations for each of these.

4.4.1 Replication

Similarly to Chapter 2, we consider component replication for the analysis in this chapter. Replication is known to be the most reliable protection technique but it comes with a high area overhead (more than double: the replicated unit plus the checker). Since the overhead of hardening/parity (larger core vulnerability) and ABFT (higher implementation overhead) solutions is expected to be higher, it makes replication a more viable alternative. Therefore we implement replication as a baseline for these techniques. We use a Dual-Modular Redundancy (DMR) approach where inputs are fed to both the original and replicated units and their outputs

are compared for consistency (Fig. 4.6). When error detection is signaled by the checker, the computation is repeated. The checking of each pair of data takes 1 cycle and is performed in parallel with computation, except for the last pair. In all of the protection configurations considered in this chapter, SRAM arrays are protected with parity, since we determined ECC to have too much overhead for small data arrays used in our platform. Parity requires retry on a detected error.

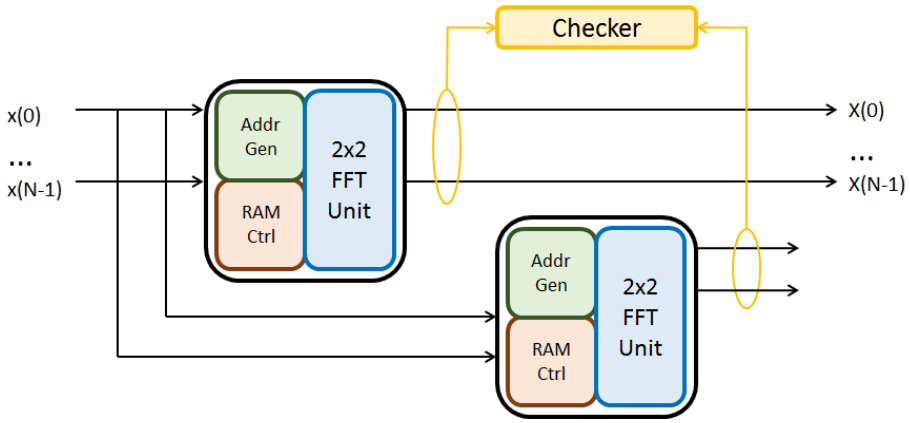

Figure 4.6: Protection of logic circuits in the FFT core with replication. The computation results from the original and the replicated core are compared.

4.4.2 Hardening

Similarly to the previous chapter, we evaluate hardening for the protection of logic circuits in the FFT accelerator. As described earlier, hardening involves adding redundant transistors to storage cells to provide redundancy that reinforces the logic state [13] [14]. Other variations implement a feedback path that cancels the effect of an upset [15]. There are a range of hardening variants that offer trade-offs in area, power and delay versus the amount of resilience provided. To achieve the best possible soft-error-rate (SER) rate, we use the heavy version [14] in our analysis, which roughly doubles the area of the protected cell. This solution is implemented at the library cell level, with no changes to the logic design. The intrinsic error cancellation capability eliminates the need for a recovery solution. Although the overhead of hardening is comparable to that of replication at the cell level, it allows selective application to only vulnerable cells, to reduce the overall core-level overhead.

4.4.3 Parity

We also consider parity for the protection of logic circuits in the FFT core. The parity encoder generates a parity signal based on the inputs to a group of flip-flops, which is then evaluated when the flip-flops are accessed [20]. At the scope of the core, error signals from different parity groups are aggregated into a single

error output for checking. The long delay of the corresponding circuitry often necessitates pipelining to avoid affecting the core's critical path, at the cost of increased detection latency. Although parity incurs a lower area overhead than hardening, it has a lower resilience capability per protected bit due to a single parity bit being used for detection. Moreover, consideration of several design parameters such as parity group size, flip-flop vulnerability, cell locality, timing slack and recovery are required to arrive at an optimal design. In the cross-layer solutions involving ABFT that we analyze in this chapter, we use hardening and parity for the protection of the remaining circuitry to reach the desired soft-error-rate (SER) target. Since the efficient flush mechanism is not available in the FFT core, we employ retry as a recovery method.

4.4.4 ABFT

The Algorithm-Based Fault Tolerance (ABFT) is a class of algorithm-specific resiliency techniques that take advantage of properties in the algorithm that relate inputs to outputs to verify correctness. The efficiency of of this approach comes from the fact that it only checks computation that makes an essential contribution to the end result in the algorithm. The ABFT solution that we develop for the FFT, based on [10], relies on the known property of the FFT stating that the product of the first input data point and the number of data points is equal to the sum of all of the output data points. The correctness of the computed output data points is therefore verified via comparing their sum to the value of the first input point.

However, from to Eq. 4.3 and 4.4, we see that any error in b, W or the multiplier will make both c and d incorrect, but the effect will not show due to the opposite signs in both c and d. To remedy this, the more robust implementation (refereed to as full ABFT in our analysis, as opposed to weak ABFT) encodes input data (with shifts and scales) to make the contribution of each output point in the sum unique. The results of calculation are then decoded (shifted and scaled back) to obtain the desired output data points (Fig. 4.7). When error detection is signaled by the checker, the computation is repeated.

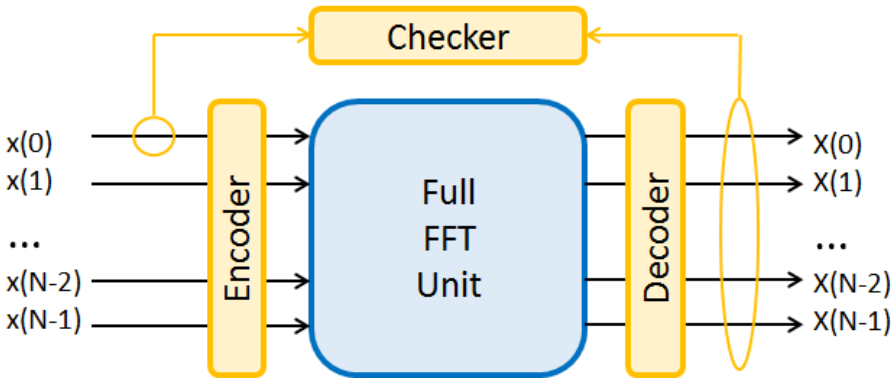

Figure 4.7: ABFT for FFT: Correctness is verified by comparing sum of output data to the first input data which involves optional encoding/decoding in a full ABFT version.

If only one encoding factor is used (a) in the form of a power of two, for simplicity, the encoder requires only one complex shift unit (two real shift units) for multiplication and one complex adder (four real adders), for the processing of a single input point, as illustrated by Eq. 4.7, where x^1 is a shifted input. The decode operation, on the other hand, then requires only one complex adder and one complex multiplier (four real multipliers and four real adders) for the processing of each output point, as illustrated in Eq. 4.8. The processing of the checksum requires one complex adder (two real adders) to process two output points.

It can be easily observed that a full implementation of a single encoder/decoder set (while 2 are needed) results in an overhead already larger than that of a full 2x2 FFT unit. This most likely implies that one such set needs to be shared across the two inputs at the cost of increasing the initial encoding and the final decoding delays. The overall size of the checking mechanism can be decreased further by reusing adders from the encoder in the decoder, since the two operations are exclusive with a small number of processing units. Finally, only two real multipliers can be used instead of four while increasing the processing time of each input to two cycles (Fig. 4.8).

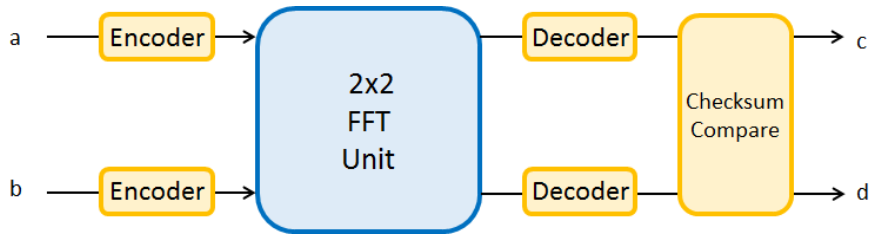

Figure 4.8: Due to large area overhead of the ABFT encoder/decoder pair, practical implementation can share one or limit its arithmetic capability at performance cost.

$$x_{encoded} = (a * x + b * x^1) \tag{4.7}$$

$$X_{decoded} = X * (a + b * W_N^k)^{-1} \tag{4.8}$$

4.5 Methodology

In this section we only briefly explain methodology for our study since it largely follows the one explained in Chapter 3.

4.5.1 Error Model

For the error model, we assume radiation-induced faults, and specifically single-event upsets (SEUs). This allows us to also explore techniques such as hardening and parity that are only applicable to this type of fault. Moreover, the transient nature of radiation-induced faults enables exploration of masking phenomenon at different levels of design hierarchy. We primary focus on the protection of logic circuits since there are existing efficient resiliency solutions for data arrays such as parity and ECC. Furthermore, we do not consider errors in combinational logic since they are shown to be much less significant [3].

4.5.2 Fault Injection

We develop a custom fault injection framework to simulate soft errors due to SEUs at the flip-flop level. The framework injects a single fault into a random flip-flop at a random cycle during a single benchmark run. The injecting RTL unit sets the appropriate mask that is then XORed with the, often multi-bit, register to inject the fault into a particular flip flop. The mask is then cleared in the following cycle and XORed with the same register to clear the fault. Because of the relatively short duration of the 1024-point input sequences, the fault injection is performed in RTL simulation. Data presented for our analysis is based on at least 10,000 injections per sequence, which allows each flip-flop to be injected at least 10 times.

4.5.3 Flip-flop Ranking for Hardening/Parity

For the application of hardening and parity, flip-flops are ranked to reflect the likelihood to propagate faults [26]. This is done via the accurate flip-flop-level injection methodology provided by Stanford University. The approach allows us to use these techniques efficiently by applying selectively to the most vulnerable flip-flops or to cover the remaining unprotected parts of circuitry for a given SER rate. Since intrinsic coverage of hardening, a cell-level technique, cannot be evaluated in RTL-level injected simulation, it is assumed based on previously conducted SPICE experiments [13].

4.5.4 Error Metrics

The outcome of the injected run is categorized as [26]: vanished, output not affected (ONA) or output mis-match (OMM) to describe its effect on the core state and user visibility. Since there is no software layer involved, unexpected termination (UT) and hang outcomes are not relevant to the accelerator. The vulnerability metric is calculated by dividing the number of occurred/unprotected errors by the total number of injections. The Soft Error Rate (SER) improvement is determined by dividing the vulnerability of the unprotected design by that of the protected design. We only consider meaningful errors (OMM) which are

useful to the design process. The touch range and detection rate are used to characterize the nature of protection.

4.5.5 Other Infrastructure

The FFT core is obtained from [21] and then extensively modified. The ABFT algorithm is implemented based on the concept presented in [10]. Analysis is conducted on a range of 10 random input sequences. We build our own tools to generate these sequences with multiple parameters, such as magnitude and repeated pattern to represent a variety inputs. The control unit implemented in the core orchestrates execution and initiates a retry run when an error is detected. Physical design overheads are evaluated using a 28nm TSMC cell library [52] and Cacti [42]. Synopsys design tools (Design Compiler, IC compiler, and PrimeTime) [53] are used to perform RTL simulation, synthesis as well as area, power and timing analysis.

4.6 Results

4.6.1 Area

As explained in Section 3, there are many ways to design an FFT processing unit with respect to the number of arithmetic units to optimize the area. Figure 4.10 (note the log scale) shows the area of the three most popular open-source architectures that we found [58]. The first architecture has half the capability of the FFT core shown in Fig. 4.9. It takes 2 cycles to process the unit work in the butterfly and requires only smaller SRAM arrays with a single read/write ports. The second architecture is a full core that processes unit work in 1 cycle while using larger SRAM arrays with dual read/write ports. The third architecture includes two FFT cores that reduce the entire computation time by half while utilizing large SRAM arrays with 4 read/write ports. The required SRAM arrays dominate the area in all architectures. Insufficient number of ports would limit the throughput of the last two architectures, particularly towards the end of computation, where data is exchanged. The dedicated arithmetic units used in the FFT cores result in an area similar to that of the general-purpose Leon core while still much smaller than the more complex IVM core, both of which were used in Chapter 3.

Figure 4.10 shows the overhead of protecting the half 2x2 FFT core with SRAM parity, different grades of ABFT and component replication. This type of core optimizes area by reducing the number of processing elements (adders and multipliers by half) while requiring an additional cycle (two total) to process unit work in the butterfly. Due to the small size of data arrays, we implement parity protection for these structures at the 3.3% overhead (encoder/decoder and data bits). SECDED ECC would be expensive in area, with 16.7%

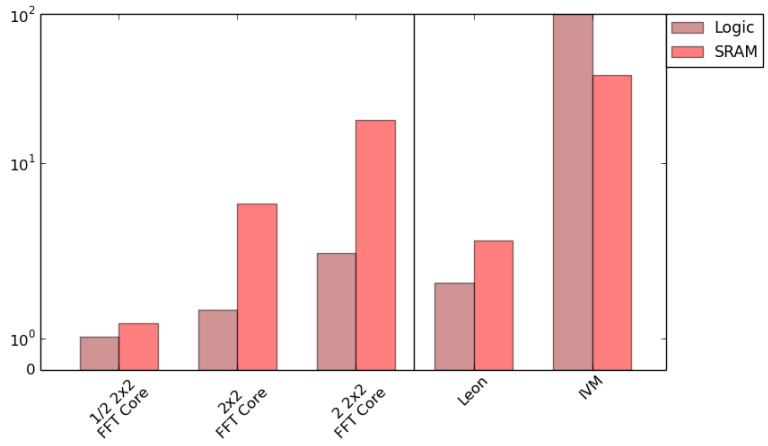

Figure 4.9: Relative area of FFT accelerator architectures compared to general-purpose cores (normalized to the largest, IVM logic). Dedicated arithmetic units and multi-port SRAM arrays increase area.

overhead, mostly due to the size of the checker. All scenarios involving ABFT require additional (hardening in this case) protection for the RAM controller, which is not covered by this technique. The weak FFT has small overhead of 5.5% due to a single adder and a comparator. To maintain the throughput of the half 2x2 FFT core, the full ABFT requires 1 encoder/decoder/checksum set which comes at a very high overhead of 69.2% due to the total of 4 multiplier and 6 adder units included. Area savings can be achieved only at the cost of reduced throughput by limiting the number of multipliers in the decoder to 2 (1/2 capability), for example, resulting in the 38.2% overhead. This is only slightly lower than replication (42.2%), which does not incur performance loss.

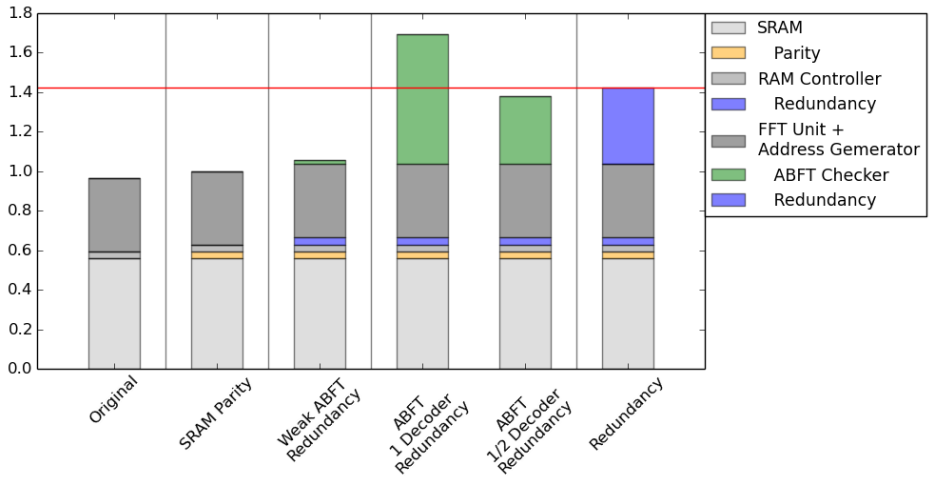

Figure 4.10: Area overhead of protection (normalized to 2nd scenario) in the half 2x2 FFT core. Area of the full ABFT implementation approaches or exceeds that of replication (indicated by the red line).

Figure 4.10 shows the overhead of protecting the full 2x2 FFT core with SRAM parity, different grades

of ABFT and component replication. We can observe the increased size of the logic, with respect to the half 2x2 core, and larger, multi-port, SRAM size with respect to logic. The overheads of weak ABFT and replication are decreased to 2.0% and 26.9%, respectively, compared to the half 2x2 core, due to the inclusion of a proportionally larger SRAM. Although the footprint of full ABFT also decreased overall, it increased with respect to logic circuits in the core. This is because 2 encoder/decoder/checksum sets are required to maintain the FFT core's throughput at a larger overhead of 48.4%. Similarly, area can be optimized only at the cost of reduced throughput by a) using only 1 decoder/checksum set (24.9%) and b) additionally reducing the number of multipliers in the decoder to 2 (1/2 capability, 13.7%). Both of these optimizations allow reducing the overhead below that of replication, which, however, does not incur performance loss. It can be inferred from Fig. 4.1 that implementations larger than 2x2 would not decrease the ABFT overhead since they would require a proportionally larger number of encoder/decoder/checksum sets. Depending on the capability of the FFT core, the number of encoder/decoder/checksum sets and read/write SRAM ports can be reduced by pipelining independent parts of the butterfly computation, while the rest of it proceeds at decreased throughput.

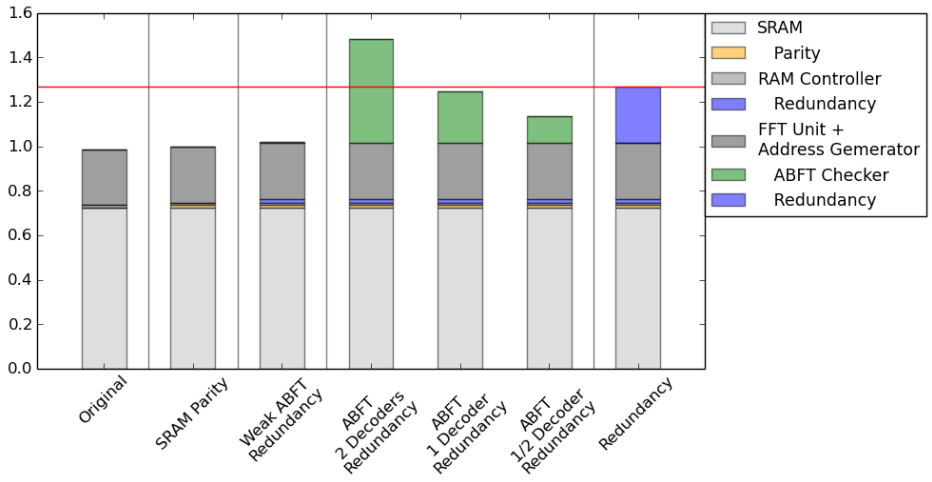

Figure 4.11: Area overhead of protection (normalized to 2nd scenario) in the full 2x2 FFT core. Incremental area optimization of full ABFT results in an increasing performance loss.

4.6.2 Performance

Since the FFT core performs the same predefined set of operations on each input, it does not exhibit performance variation. This is unlike the general-purpose core, where the execution depends on the control flow in the benchmark. Assuming that our accelerator reads inputs already present in memory, there is no initial delay while waiting for the first pair of data in the ordered sequence. Similarly, there is no final delay

due to the ordering of results, since they are written to respective memory locations as they become available. The half 2x2 FFT core reads 1 input per cycle and takes 2 cycles to process unit work in the butterfly (Fig. 4.4), where a single-port array provides enough bandwidth. In the case of the full 2x2 FFT core, 2 inputs are read per cycle to complete computation in 1 cycle, where two-port arrays are needed. All except for the first read and last write memory access are overlapped with computation. The half 2x2 core, for example, takes 10,240+2 cycles to process the 1024-point long input sequence (Fig. 4.12, bar 1 from top). In the case of the full 2x2 FFT core or the one with 2 2x2 units, this duration is decreased to 5,120+2 (bar 3)and 2,560+2 cycles respectively.

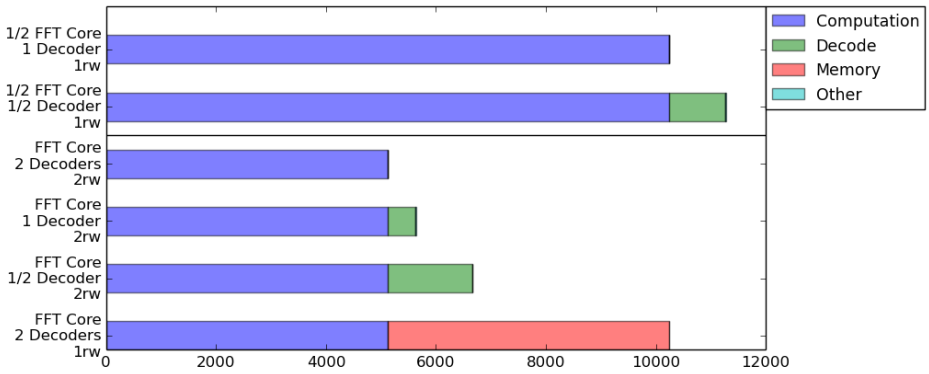

Figure 4.12: Performance overhead due to area optimization in the decoder (less multipliers) and SRAM arrays (single port) for the half and full 2x2 FFT cores.

In the case of component replication, all except for the last result are checked (compared) in parallel with computation, in the cycle following their creation, when results are written to memory. Both weak and full ABFT require an additional final cycle to compare the checksum. In addition, full ABFT needs 2 more, initial and final, cycles to insert encode and decode operations. The ABFT area optimizations that rely on a) limiting the number of decoder/comparator sets or b) the number of multipliers in the decoder result increase the decoding latency at the back end of computation. In this case, only part of the decoding process can be overlapped with computation. In the case of the half 2x2 FFT core, reducing the complexity of the decoder adds 1024 cycles while causing ∼10% performance overhead (bar 2). In the case of the full 2x2 FFT core, employing only 1 decoder/checksum set adds 512 cycles while causing a similar ∼10% overhead. Further area optimization by reducing the number of multipliers in the decoder results in 1536 additional cycles and ∼30% performance loss. Strictly form the area-performance point of view, the slight advantage of ABFT over replication in terms of area (4-2%, Fig. 4.10 and 4.11) may not justify the ∼10% loss in performance. The performance overhead would be smaller if more complex computation, such as that in a 2D FFT, were involved.

4.6.3 FFT Core Vulnerability

The fault injection results obtained from the half 2x2 FFT core show most of the injected faults (68.1%) resulting in output mis-match (OMM) errors that are visible in the program output (Fig. 4.13). This number should be somewhat smaller in the case of the full 2x2 FFT core that has higher utilization due to performing all computation in a single cycle. Unlike in the general purpose core, the FFT cores implement a fixed control flow and it is therefore not susceptible to unexpected termination (UT) and crash outcomes that are typically related to faults in control flow instructions. The second largest group are vanished errors (21.0%) that neither affect the accelerator state nor the program output. The smallest group are output-not-affected (ONA) errors (10.8%) that have an effect on the architectural state but not the program output. While many of the bits often receive an error on a fault (depending on implementation), others are less susceptible due to partial component utilization and logical masking that depends on the processed data (mostly those in the FFT unit). Therefore, depending on the time of injection, some faults might end up in any of the three error categories. The number of OMM errors is significantly larger than that for the general-purpose cores studied in Chapter 3. Since the FFT core is designed for a particular algorithm, most of the features are utilized in the computation and essential to the correctness of the result. The vulnerability in general-purpose cores, on the other hand, depends on the program flow, instructions (e.g., proportion of shift, arithmetic vs. boolean, floating-point, etc.), unit functionality (essential or performance-only) and type of executed code (user, operating system or library calls).

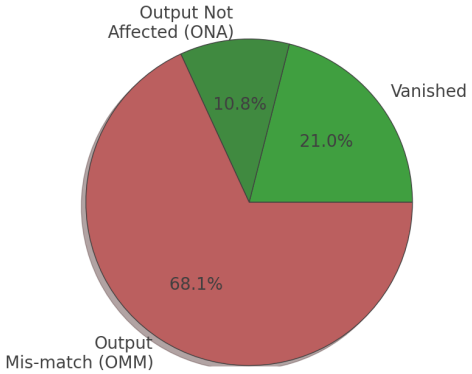

Figure 4.13: Distribution of error types (effect on the core state and program output) resulting from injected faults in the 2x2 FFT core. Most faults result in meaningful errors.

Figure 4.14 shows the breakdown of meaningful errors (OMM) and vulnerable flip-flops across FFT core components. Due to relative component sizes, most injected faults (and therefore vulnerable flip-flops) occur in the FFT unit, while smaller amounts of errors take place in the remaining components (address generator and RAM controller). We can observe that in the address generator, a large fraction of flip-flops (82.4%) are

vulnerable (receive an error at some point of time) while 73.1% of errors injected in that flip-flop range are meaningful (affect the output). This unit is fairly sensitive, since it controls signals for the orchestration of the unit and butterfly FFT computation. While the FFT unit also has a large range of vulnerable flip-flops (88.6%), it receives fewer meaningful errors (62.4%). This is because not all arithmetic units are utilized every cycle in the half 2x2 unit, and there is some degree of logical masking depending on the processed data. Finally, the RAM controller appears to be proportionally the most vulnerable unit with 100% vulnerable flip flops and 91.0% meaningful error rate. The unit is very sensitive to faults that could result in a wrong read/write access to a wrong input/output real/imaginary data array during any cycle. Moreover, the FFT core includes a few flip-flops that are used (and therefore not removed in synthesis) where the effect of errors is always masked.

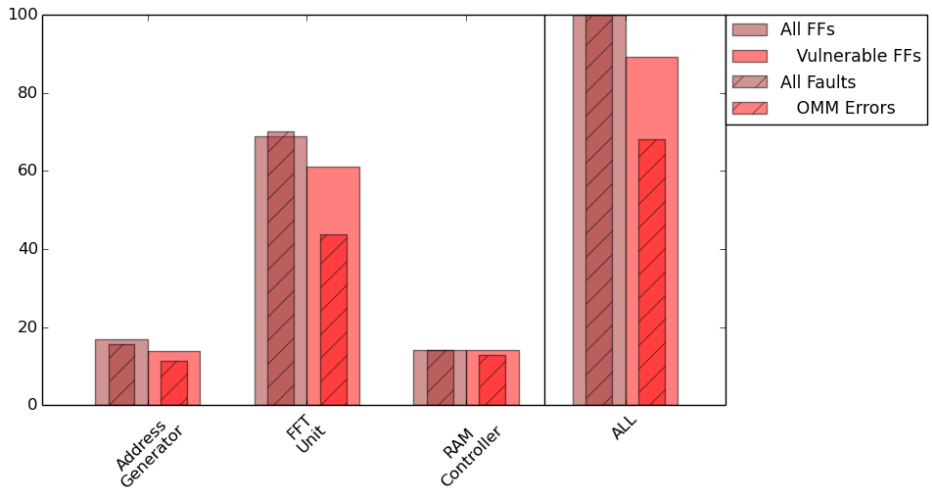

Figure 4.14: Distribution of all/vulnerable flip-flops and all/meaningful injected faults in FFT core components in the half 2x2 FFT core (percentage). The largest unit (FFT) receives most errors.

4.6.4 Single-Layer Protection

Figure 4.15 shows the breakdown of detected meaningful errors (OMM) and protected vulnerable flip-flops across FFT core components for the considered resiliency techniques. Since the replication approach checks every element in the output, it detects every injected fault (except for a very rare case, discussed later) and therefore protects all vulnerable flip-flops. The ABFT detection, on the other hand, is based on a checksum of all output elements. It will therefore not be able to detect faults that cause errors in multiple output elements that cancel out in the checksum. While replication can be applied to all logic units, ABFT can only protect the FFT computation (address generator and the FFT unit), while requiring additional protection for the RAM controller. As described earlier, detected meaningful errors can originate from different flip flops and

cycles. Therefore the range of (partially) protected (or "touched") flip-flops is wider than the corresponding detection rate for all units in Fig. 4.15. This effect is not so prominent in accelerators, where errors are more confined to a fixed range of flip-flops.

In the case of the FFT unit, our injection results show that the weak ABFT detects less than half of the injected errors (37.12%) and protects the corresponding 45.1% of vulnerable flip-flips, which confirms analysis in [10]. Because of the specific data flow in FFT, explained in Section 3, errors in the (data or weight) inputs to the multiplier, including the multiplier itself, can change the value of a pair of outputs while leaving the checksum unaffected due to the cancellation effect. Since more than half of the flip-flops are susceptible to this effect, weak FFT is unable to protect them. Full FFT, on the other hand, detects 91.6% of all meaningful errors and protects 96.1% of vulnerable flip-flops in the FFT unit by avoiding the cancellation effect via encoding/decoding. Since the FFT checker uses weights generated by the FFT unit, we discovered that faults in the relevant data can cause the decoder to mask the error originating in the FFT core, which accounts for the few undetected errors remaining there. Full ABFT has a high error and flip-flop coverage (80.4% and 94.2%, respectively) in the address generator. We also observed that some faults in this unit cause data to be computed and written out in the wrong order while not affecting the checksum, which accounts for most of the undetected errors there.

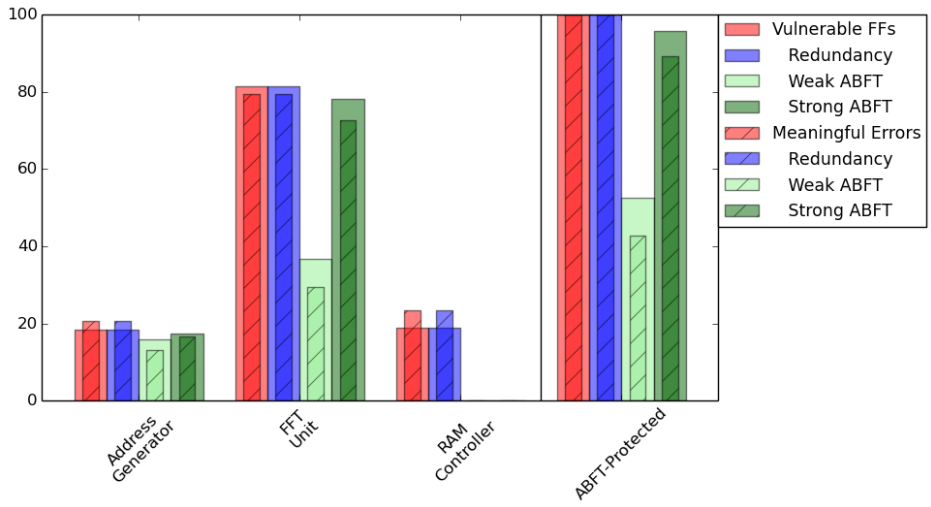

Figure 4.15: Distribution of all/covered vulnerable flip-flops and all/detected meaningful errors in FFT core components (percentage). Full ABFT more than doubles coverage of weak ABFT.

Figure 4.16 shows the vulnerability distribution for all flip-flops in the baseline unprotected design as well as the design protected by weak and full ABFT (note the log scale). We can observe that weak ABFT decreases the error rate for many of the vulnerable flip flops (moves them from the more to less vulnerable range), while full ABFT further improves that metric by providing close to full protection for most of them.

Weak and full ABFT improve the overall soft error rate (SER) by 1.55x and 13.76x, respectively, compared to replication which provides complete protection (except for a very rare case, discussed below). As mentioned in the methodology section, the SER improvement is a sum of vulnerability improvements across all flip-flop. This data gives a sense of what range of flip-flips are "touched" and perhaps considered (partially) protected by the various techniques, while the additional complementary protection can be used for the remaining circuitry. Although the error detection rate is high for a corresponding range of "touched" flip-flops in the full ABFT, there are still some unprotected flip-flops that cause errors not detectable with this technique. These may require additional protection to reach higher resilience targets.

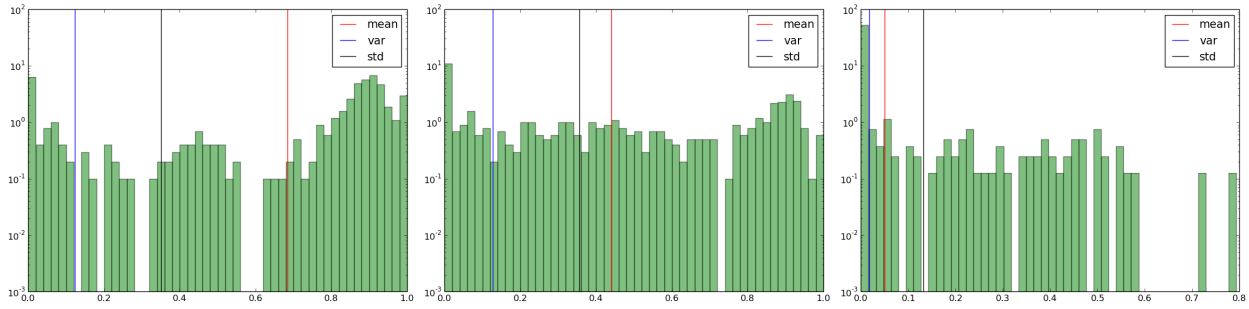

Figure 4.16: Flip-flop vulnerability distribution for a) unprotected, b) weak ABFT-protected and c) full ABFT-protected design (percentage, log scale). Highly vulnerable flip-flops are reduced by b) and c).

4.6.5 Cross-Layer Protection

The cross-layer resilience approach presented in this dissertation, assumes evaluation of protection techniques according to their best fit and complementing them to reach desired soft error rate (SER) improvement. In Fig. 4.10 and 4.11 we already complemented ABFT with replication for the protection of the RAM controller to show the upper-bound area overhead. Similarly to Chapter 3, we now use the hardening/parity techniques as a complementary solution to reach a set of SER targets since it can achieve lower overhead via selective fine-grained application.

Fig. 4.17 shows the protection overhead for the half 2x2 FFT core. Although the full ABFT implementation (with a single decoder/checksum set to match the core's throughput) requires little augmentation from hardening/parity to reach high SER targets, its checker has a very large overhead, higher than that of replication. The reduction of the decoder complexity allows decreasing this overhead below that of replication at the ∼10% performance penalty. The same comprehensive coverage can be achieved at a much lower overhead with hardening/parity. This solution has a much better efficiency, since it directly protects vulnerable flip-flops, which does not involve replication of all circuitry or use of expensive functional units, such as in replication or ABFT, respectively. It can be observed that achieving 30x SER improvement already involves

gradually covering almost all vulnerable flip-flops in the core, requiring no more coverage for higher SER targets. However, the most interesting result is the combination of the weak ABFT and hardening/parity, which incurs even lower area overhead for almost all SER targets. The weak ABFT provides a reasonably large flip-flop coverage (52.7%) and error detection (42.7%) at a very small 5.5% area overhead. This efficient coverage provided by weak ABFT requires only an insignificant amount of augmentation with hardening/parity. This results in the lowest overall area overhead at an insignificant performance penalty (2 cycles for addition and comparison).

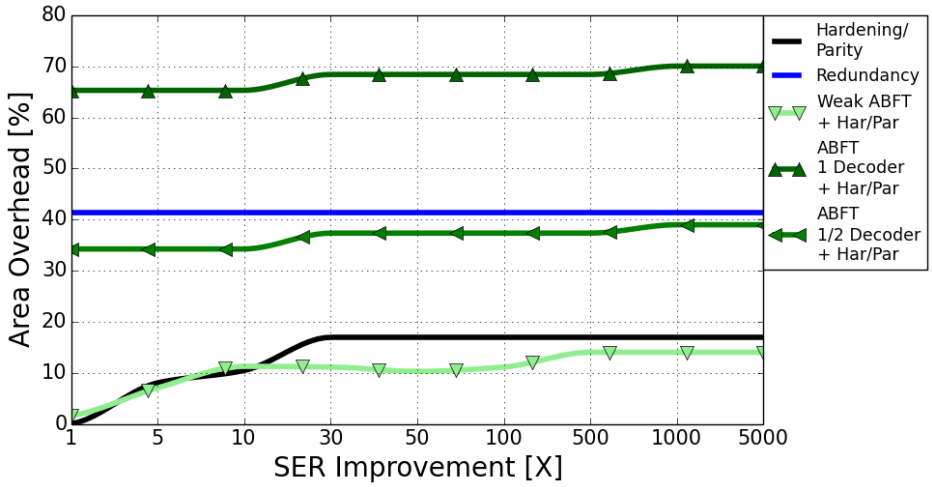

Figure 4.17: Area overhead of cross-layer protection in the half 2x2 FFT core. The combination of weak ABFT and hardening/parity is the most area and performance optimal.

Fig. 4.17 shows the protection overhead for the full 2x2 FFT core. This core variant features larger logic, dual-port SRAM arrays and 2 ABFT encoder/decoder/checksum sets. As a result, the area overhead of the full ABFT implementation is even larger with respect to the replication approach. Similarly to the half 2x2 FFT core, a combination of weak ABFT and hardening/parity proves to be the most efficient solution. We can also observe that the use of a single simplified decoder (reduced number of multipliers from 4 to 2) in the full ABFT allows achieving overhead closer to that of hardening/parity at a ∼30% performance penalty. Further area optimization to match that achievable with a combination of weak ABFT and hardening/parity is possible when using a single multiplier in the decoder at a significant performance penalty of 70%. These results prove full ABFT to be impractical for the hardware FFT implementation due to a large checker overhead, for both half 2x2 and full 2x2 FFT cores. This also holds for systems that involve more FFT units where the overhead is expected to be similar to that of the full 2x2 FFT core due to additional encoder/decoder/checksum sets involved. Therefore, as illustrated by our results only the weak version of ABFT is feasible, and it provides optimal coverage when combined with low-level hardware

hardening/parity.

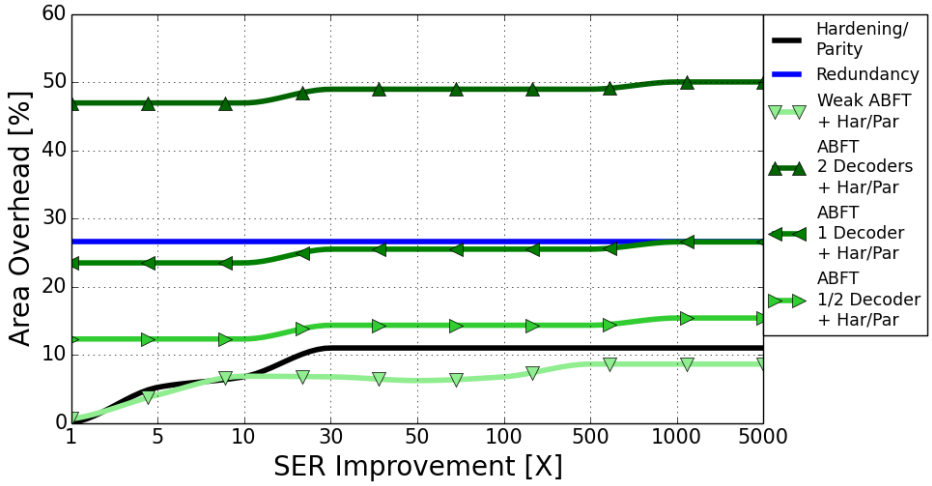

Figure 4.18: Area overhead of cross-layer protection in the full 2x2 FFT core. The full ABFT implementation is even less area and performance optimal for larger system.

4.6.6 Workload Dependence

In the general-purpose processor, benchmark characteristics have a more significant effect on the vulnerability of the processor in the particular execution run. This is because benchmarks includes different types of instructions as well as varying durations of compute and memory access cycles. In our accelerator core, these three factors are fixed and the only minor vulnerability variation across runs comes from logical masking due to different data processed. However, these differences cancel out in the sum of different sequences simulated in our analysis. Therefore our results represent core vulnerability for an average case.

4.6.7 Relevance to Other Accelerators

Since resilience solutions such as hardening and replication protect particular discrete features in the core, it is relatively easier to determine their applicability and overhead for any accelerator. However, it is more challenging to reason about the specific ABFT approach that is somewhat different for every special-purpose unit, if applicable. We selected FFT as an example accelerator for our experiments because it is a widely used, sophisticated algorithm with a significant amount of data sharing that increases vulnerability. Moreover, as mentioned earlier, it involves many trade-offs with respect to the processing unit architecture. Finally, FFT has two versions of the ABFT checker that represent extreme ends of the overhead spectrum that can easily encompass other accelerators. Since most other algorithms, such as matrix multiply and other arithmetic operations, calculate a checksum for ranges of values [18], they require more computation than a

simple checksum in the weak ABFT, while significantly less than encoding/decoding in the full ABFT. The weak ABFT, complemented with hardening/parity, provides a benefit because of its small area compared to the complex 2x2 FFT unit and the corresponding hardening/parity protection overhead. Therefore, even this efficient approach may not be competitive against hardening/parity in the case of algorithms with less sophisticated hardware implementations and relatively larger checker sizes. However, further investigation is required to explore particular differences between accelerators and their specialized protection. Since accelerator architectures vary, the overall overheads also depend on the proportions of vulnerable logic and parity/ECC-protected SRAM circuits.

4.6.8 Protection of the Checker

We also evaluate the vulnerability of the checker for a) a single fault in the system and b) two faults affecting the FFT unit and the checker. In both cases, almost all errors in the ABFT (encoder, decoder, checksum, comparator) and replication (redundant unit, comparator) checkers cause a discrepancy in the result or the checksum that aligns with the originally detected error or causes a new false-positive detection. The false-positive detection is only a performance concern since it only unnecessarily triggers recovery via recomputation. The occurrence of a false-negative detection is a real concern, but is however very unlikely. It would either require a single fault that flips the error status bit or two faults that affect particular data at specific times to cause cancellation (probability approaching 0, given low error rate and the number of involved flip-flops). The latter is even less likely in the case of replication that checks every individual result. Under fault injection testing, we show that around 97%/93%/89% of flip flops in the weak-ABFT/full-ABFT/Replication checkers in the half 2x2 FFT core are vulnerable, with 82%/74%/68% of injected faults causing errors. However, among those, only 0.76%, 0.11% and 0.12% cause false-negative detection that affects the correctness of the results. The remaining errors cause false-positive detection that only unnecessary triggers retry at a performance loss. However, due to the much smaller area, the weak ABFT checker experiences a proportionally smaller real-world error rate than the full ABFT or replication. Unnecessary performance loss could be avoided by protecting the checker with hardening/parity, for example, at an additional area overhead, analogous to that in the FFT core analyzed earlier.

4.7 Conclusions and Future Work

In this chapter, we performed a detailed analysis of the resilience solutions for the Fast Fourier Transform (FFT) core, an example accelerator, with the particular focus on the Algorithm-Specific Fault Tolerance (ABFT) approach. The first goal of our work was to determine the vulnerability of the FFT core to illustrate

relative differences from general-purpose cores, studied in previous chapter, which have an effect on applicable resilience solutions. Achieving this goal required obtaining an open-source FFT core as well as making significant modifications to improve modularity. We developed a custom injection framework to perform vulnerability analysis for a range of input sequences. With our results, we illustrate that the accelerator receives a significantly larger number of meaningful errors compared to general-purpose cores. This is because, in a special-purpose design, the overall hardware utilization is higher. Due to the same reason, meaningful errors are more confined to a particular range of vulnerable flip-flops, making the applied resilience more cost-effective.

Another goal of our work was to show the effectiveness of different resilience techniques in the context of our example FFT accelerator. Achieving this goal required the implementation of hardening/parity, replication, and two grades of ABFT for our example FFT core. While hardening/parity, weak ABFT, and replication have deterministic overheads, the overhead of full ABFT varies depending on the number of the encoding/decoding circuits required to match the throughput of various FFT core architectures. Our results show that full ABFT doubles the coverage of weak ABFT while approaching that of replication. However, in spite of area optimizations at the cost of performance, the overhead of full ABFT cannot be significantly decreased. The most area and performance-optimal protection is achieved with a combination of weak ABFT and hardening/parity. In the case of other special-purpose cores, where size of the checker can be relatively larger compared to the processing unit itself, the hardening/parity approach is expected to be more beneficial than ABFT. The future work can focus on similar analysis performed on other popular accelerators to explore relative differences as well as reliability comparisons against the same functionality in the general-purpose core.

Chapter 5

Conclusions and Future Work

5.1 Dissertation Summary

The soft-error problem remains a significant design concern due to the continuing scaling of devices (in spite of recent advances in fabrication that decrease this effect), the increasing complexity of processors (or entire systems, such as supercomputers) and a variety of sensitive applications and environments (automotive, aviation). As a result of this, the overhead of required protection increases to provide the same or better level of resilience. It is therefore becoming necessary to find optimal ways to achieve resilience with the existing techniques. This inevitably requires considering a larger number of techniques at different (hardware, software, architecture) layers of system design.

However, there are several challenges that need to be faced before arriving at a solution to the above problem. First, the overhead of various protection techniques needs to be understood for different types of circuits and components. It is important to gain more insights into the more detailed breakdown of these overheads and ways of optimizing the overall core-level resilience. Second, it is necessary to explore more efficient resilience solutions that involve techniques from difference layers of system design as well as their complementary combinations. This should allow achieving the desired error reduction at the lowest area, power and performance overhead. Finally, similar analysis should be performed for a special-purpose architecture to observe relative differences.

To address these concerns, this dissertation endeavors to understand and optimize soft-error protection with combinations of existing and proposed resilience solutions. We use a simple and a complex general-purpose core as well as an example accelerator to make our results more generally applicable. A summary of this dissertation and its contributions are as follows.

1. In Chapter 2, we develop a framework to provide an analysis of various protection choices to improve our understanding of their applicability as well as their sources of overheads at component and core-level granularities. We first show how the protection overhead varies depending on the amount of vulnerable features (flip-flops) and the applicability of protection techniques. Subsequently, we break down the checker structure to illustrate, among others, the significant overhead of ECC for logic circuits. Finally we explore interesting hybrid resilience combinations for comprehensive coverage aimed at improving multiple design metrics.

2. In Chapter 3, we implement Data-flow Checking (DFC), an architecture-level resilience technique, and analyze its benefits in combination with low-overhead hardware and software solutions in the cross-layer approach. We first illustrate coverage differences of DFC resulting from design choices and underlying architectures. Subsequently, we determine DFC to be less competitive in coverage, performance and area (especially for the small Leon core with recovery enabled) than a combination of hardening and parity applied selectively at a flip-flop level. Finally, we show that as opposed to ABFT, DFC yields no benefit in cross-layer combinations.

3. In Chapter 4, we develop and analyze a similar set of protection techniques, with a particular focus on Algorithm-based Fault Tolerance (ABFT), for the FFT accelerator. We first study different configurations of the accelerator core and its protection at varying overheads. Subsequently, we show that, compared to the general purpose core, the vulnerability of the accelerator and the efficiency of ABFT are higher due to the proportionally larger amount of core features affecting the output. Finally, due to the large overhead of ABFT, we illustrate that only its simple version combined with hardening/parity provides optimal protection.

5.2 Future Research Challenges and Directions

Our experiences described in Chapter 3 illustrate the benefit of the cross-layer analysis in the optimal resilience design process. The resilience design process would also benefit from extending to a wider variety of solutions. The design scenarios that we evaluate in our work represent more competitive alternatives to traditionally used solutions, in terms of a combination of area, power and performance metrics. However, there are other resilience choices which may optimize an individual metric, which we did not consider for further study. Among others, these include Razor latches [29], Redundant Multi-Threading (RMT) [35], monitor cores [30], symptom-based detection [60] and approximate resilience [61]. Moreover, fine-grained flip-flop level fault injection capability is not always available to designers, in which case less-accurate architecture-level injection

in the simulator is used. It is important to reconcile the two approaches and be able to improve the efficiency of the resilient design based on data obtained from each.

The cross-layer analysis approach would be even more useful to the designer if it were more generic and automated, much like in our work presented in Chapter 2. The application of hardening, parity and replication to logic circuits as well as parity and ECC to SRAM arrays is straightforward and can be easily automated for any input RTL code. The evaluation of ABFT, EDDI and CFCSS is already possible for various architectures with the ABFT library that we created and compiler tools available for other techniques. The corresponding hardware-level fault injection would need to be performed via generic tools from Synopsis or Cadence, rather than our custom RTL injectors. Without manual implementation, the overhead or DFC and other watchdog-type checking could only be estimated, while the testing of its detection capability would require fault injection in a generic (or specific, if available) simulator. Such injection capability in a generic simulator for DFC, and optionally ABFT, would still need to be developed.

Analysis of a wider range of hardware ABFT techniques in accelerators would require manual implementation due to specifics of each solution. This process can be more automated if the underlying hardware is created from a high-level description, such as in [62], which then needs to be optimized. Another related area that requires exploration is resilience in Graphics Processing Units (GPUs). While these architectures are similar to general-purpose processors in generic workloads executed, they include a much larger number of processing units capable of large-scale computation that is even more vulnerable to soft-errors. So far, only ECC protection for memory [63], variants of workload replication at different levels (number or size of workgroups) [64] and various program checking [65] have been explored for GPUs. It is therefore important to analyze alternative hardware approaches such as hardening, parity, replication as well as flow checking and monitor cores. The resilience of GPUs is becoming more important as they are more often used in supercomputers. In this case, errors can be especially important because they can accumulate across iterations while many applications require high precision.

Bibliography

[1] R.C. Baumann. Radiation-induced soft errors in advanced semiconductor technologies. *Device and Materials Reliability, IEEE Transactions on*, 5(3):305–316, Sept 2005.

[2] S. Natarajan, M.A. Breuer, and S.K. Gupta. Process variations and their impact on circuit operation. In *Defect and Fault Tolerance in VLSI Systems, 1998. Proceedings., 1998 IEEE International Symposium on*, pages 73–81, Nov 1998.

[3] N. Seifert, B. Gill, S. Jahinuzzaman, J. Basile, V. Ambrose, Quan Shi, R. Allmon, and A. Bramnik. Soft error susceptibilities of 22 nm tri-gate devices. *Nuclear Science, IEEE Transactions on*, 59(6):2666–2673, Dec 2012.

[4] E.H. Cannon, D.D. Reinhardt, M.S. Gordon, and P.S. Makowenskyj. Sram ser in 90, 130 and 180 nm bulk and soi technologies. In *Reliability Physics Symposium Proceedings, 2004. 42nd Annual. 2004 IEEE International*, pages 300–304, April 2004.

[5] P.J. Meaney, S.B. Swaney, P.N. Sanda, and L. Spainhower. Ibm z990 soft error detection and recovery. *Device and Materials Reliability, IEEE Transactions on*, 5(3):419–427, Sept 2005.

[6] D. Burgess, E. Gieske, J. Holt, T. Hoy, and G. Whisenhunt. e6500: Freescale's low-power, high-performance multithreaded embedded processor. *Micro, IEEE*, 32(5):26–36, Sept 2012.

[7] Rajshekar Kalayappan and Smruti R. Sarangi. A survey of checker architectures. *ACM Comput. Surv.*, 45(4):48:1–48:34, August 2013.

[8] N.J. George, C.R. Elks, B.W. Johnson, and J. Lach. Bit-slice logic interleaving for spatial multi-bit soft-error tolerance. In *Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on*, pages 141–150, June 2010.

[9] Quora. Fft computation diagram, 2013.

[10] J.-Y. Jou and J.A. Abraham. Fault-tolerant fft networks. *Computers, IEEE Transactions on*, 37(5):548–561, May 1988.

[11] S. Herbert and D. Marculescu. Analysis of dynamic voltage/frequency scaling in chip-multiprocessors. In *Low Power Electronics and Design (ISLPED), 2007 ACM/IEEE International Symposium on*, pages 38–43, Aug 2007.

[12] Shubu Mukherjee. *Architecture Design for Soft Errors*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[13] T. Calin, M. Nicolaidis, and R. Velazco. Upset hardened memory design for submicron cmos technology. *Nuclear Science, IEEE Transactions on*, 43(6):2874–2878, Dec 1996.

[14] Ming Zhang, S Mitra, T. M. Mak, N. Seifert, N.J. Wang, Quan Shi, Kee Sup Kim, N.R. Shanbhag, and S.J. Patel. Sequential element design with built-in soft error resilience. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 14(12):1368–1378, Dec 2006.

[15] Hsiao-Heng Kelin Lee, K. Lilja, M. Bounasser, P. Relangi, I.R. Linscott, U.S. Inan, and S. Mitra. Leap: Layout design through error-aware transistor positioning for soft-error resilient sequential cell design. In *Reliability Physics Symposium (IRPS), 2010 IEEE International*, pages 203–212, May 2010.

[16] M.A. Schuette and J.P. Shen. Processor control flow monitoring using signatured instruction streams. *Computers, IEEE Transactions on*, C-36(3):264–276, March 1987.

[17] A. Meixner, M.E. Bauer, and D.J. Sorin. Argus: Low-cost, comprehensive error detection in simple cores. In *Microarchitecture, 2007. MICRO 2007. 40th Annual IEEE/ACM International Symposium on*, pages 210–222, Dec 2007.

[18] Kuang-Hua Huang and J.A. Abraham. Algorithm-based fault tolerance for matrix operations. *Computers, IEEE Transactions on*, C-33(6):518–528, June 1984.

[19] D. Lipetz and E. Schwarz. Self checking in current floating-point units. In *Computer Arithmetic (ARITH), 2011 20th IEEE Symposium on*, pages 73–76, July 2011.

[20] S Mitra and E.J. McCluskey. Which concurrent error detection scheme to choose ? In *Test Conference, 2000. Proceedings. International*, pages 985–994, 2000.

[21] Digital Filter. Fft laboratory, 2014.

[22] S.S. Mukherjee, J. Emer, and S.K. Reinhardt. The soft error problem: an architectural perspective. In *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*, pages 243–247, Feb 2005.

[23] L.G. Szafaryn, B.H. Meyer, and K. Skadron. Evaluating overheads of multibit soft-error protection in the processor core. *Micro, IEEE*, 33(4):56–65, July 2013.

[24] OpenCores. Openrisc processor, 2011.

[25] A. Heinecke, M. Klemm, and H.-J. Bungartz. From gpgpu to many-core: Nvidia fermi and intel many integrated core architecture. *Computing in Science Engineering*, 14(2):78–83, March 2012.

[26] Hyungmin Cho, S. Mirkhani, Chen-Yong Cher, J.A. Abraham, and S. Mitra. Quantitative evaluation of soft error injection techniques for robust system design. In *Design Automation Conference (DAC), 2013 50th ACM / EDAC / IEEE*, pages 1–10, May 2013.

[27] N. Seifert and M. Kirsch. Real-time soft-error testing results of 45-nm, high-k metal gate, bulk cmos srams. *Nuclear Science, IEEE Transactions on*, 59(6):2818–2823, Dec 2012.

[28] J.A. Rivers, M.S. Gupta, J. Shin, P.N. Kudva, and P. Bose. Error tolerance in server class processors. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 30(7):945–959, July 2011.

[29] D. Ernst, Nam Sung Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, and T. Mudge. Razor: a low-power pipeline based on circuit-level timing speculation. In *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*, pages 7–18, Dec 2003.

[30] T.M. Austin. Diva: a reliable substrate for deep submicron microarchitecture design. In *Microarchitecture, 1999. MICRO-32. Proceedings. 32nd Annual International Symposium on*, pages 196–207, 1999.

[31] A. Meixner and D.J. Sorin. Error detection using dynamic dataflow verification. In *Parallel Architecture and Compilation Techniques, 2007. PACT 2007. 16th International Conference on*, pages 104–118, Sept 2007.

[32] N. Oh, P.P. Shirvani, and E.J. McCluskey. Control-flow checking by software signatures. *Reliability, IEEE Transactions on*, 51(1):111–122, Mar 2002.

[33] N. Oh, P.P. Shirvani, and E.J. McCluskey. Error detection by duplicated instructions in super-scalar processors. *Reliability, IEEE Transactions on*, 51(1):63–75, Mar 2002.

[34] George A. Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, and David I. August. Swift: Software implemented fault tolerance. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '05, pages 243–254, Washington, DC, USA, 2005. IEEE Computer Society.

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

[36] N.J. Wang and S.J. Patel. Restore: Symptom-based soft error detection in microprocessors. *Dependable and Secure Computing, IEEE Transactions on*, 3(3):188–201, July 2006.

[37] Jangwoo Kim, Nikos Hardavellas, Ken Mai, Babak Falsafi, and James Hoe. Multi-bit error tolerant caches using two-dimensional error coding. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, pages 197–209, Washington, DC, USA, 2007. IEEE Computer Society.

[38] Jared C. Smolens, Brian T. Gold, Jangwoo Kim, Babak Falsafi, James C. Hoe, and Andreas G. Nowatzyk. Fingerprinting: Bounding soft-error detection latency and bandwidth. *SIGARCH Comput. Archit. News*, 32(5):224–234, October 2004.

[39] Cadence. Encounter rtl compiler, 2014.

[40] John L. Henning. Spec cpu2000: measuring cpu performance in the new millennium. *Computer*, 33(7):28–35, Jul 2000.

[41] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011.

[42] S.J.E. Wilton and N.P. Jouppi. Cacti: an enhanced cache access and cycle time model. *Solid-State Circuits, IEEE Journal of*, 31(5):677–688, May 1996.

[43] S. Mitra, P. Bose, E. Cheng, C.-Y. Cher, H. Cho, R. Joshi, Y.M. Kim, C.R. Lefurgy, Y. Li, K.P. Rodbell, K. Skadron, J. Stathis, and L. Szafaryn. The resilience wall: Cross-layer solution strategies. In *VLSI Technology, Systems and Application (VLSI-TSA), Proceedings of Technical Program - 2014 International Symposium on*, pages 1–11, April 2014.

[44] B. Stackhouse, S. Bhimji, C. Bostak, D. Bradley, B. Cherkauer, J. Desai, E. Francom, M. Gowan, P. Gronowski, D. Krueger, C. Morganti, and S. Troyer. A 65 nm 2-billion transistor quad-core itanium processor. *Solid-State Circuits, IEEE Journal of*, 44(1):18–31, Jan 2009.

[45] M. Tsunoyama and S. Naito. A fault-tolerant fft processor. In *Fault-Tolerant Computing, 1991. FTCS-21. Digest of Papers., Twenty-First International Symposium*, pages 128–135, June 1991.

[46] Shubhendu S. Mukherjee, Christopher Weaver, Joel Emer, Steven K. Reinhardt, and Todd Austin. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, pages 29–, Washington, DC, USA, 2003. IEEE Computer Society.

[47] Gaisler Research. Leon3 processor, 2015.

[48] 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 *Dependable Systems and Networks, 2004 International Conference on*, pages 61–70, June 2004.

[49] Eli Bendersky. Python elf tools, 2014.

[50] John D Davis, Charles P Thacker, Chen Chang, C Thacker, and J Davis. Bee3: Revitalizing computer architecture research. *Microsoft Research*, 2009.

[51] Texas Advanced Computing Center. Stampede, 2015.

[52] Taiwan Semiconductor Manufacturing Company. 28nm library, 2015.

[53] Synopsys. Design tools, 2015.

[54] S. Borkar. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. *Micro, IEEE*, 25(6):10–16, Nov 2005.

[55] Yi-Pin Fang and A.S. Oates. Muon-induced soft errors in sram circuits in the terrestrial environment. *Device and Materials Reliability, IEEE Transactions on*, 15(1):115–122, March 2015.

[56] Shahrzad Mirkhani, Subhasish Mitra, Chen-Yong Cher, and Jacob Abraham. Efficient soft error vulnerability estimation of complex designs. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, pages 103–108. EDA Consortium, 2015.

[57] Wikipedia. Intel 8087, 2014.

[58] OpenCores. Projects, 2015.

[59] Liang Wang and K. Skadron. Implications of the power wall: Dim cores and reconfigurable logic. *Micro, IEEE*, 33(5):40–48, Sept 2013.

[60] S.K. Sahoo, Man-Lap Li, P. Ramachandran, S.V. Adve, V.S. Adve, and Yuanyuan Zhou. Using likely program invariants to detect hardware errors. In *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*, pages 70–79, June 2008.

[61] Vinay K. Chippa, Srimat T. Chakradhar, Kaushik Roy, and Anand Raghunathan. Analysis and characterization of inherent application resilience for approximate computing. In *Proceedings of the 50th Annual Design Automation Conference*, DAC '13, pages 113:1–113:9, New York, NY, USA, 2013. ACM.

[62] Y.S. Shao, B. Reagen, Gu-Yeon Wei, and D. Brooks. Aladdin: A pre-rtl, power-performance accelerator simulator enabling large design space exploration of customized architectures. In *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*, pages 97–108, June 2014.

[63] Bo Fang, Jiesheng Wei, K. Pattabiraman, and M. Ripeanu. Abstract: Evaluating error resiliency of gpgpu applications. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 1502–1503, Nov 2012.

[64] J. Wadden, A. Lyashevsky, S. Gurumurthi, V. Sridharan, and K. Skadron. Real-world design and evaluation of compiler-managed gpu redundant multithreading. In *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*, pages 73–84, June 2014.

[65] Si Li, Naila Farooqui, and Sudhakar Yalamanchili. Software reliability enhancements for gpu applications. In *Proceedings of the Sixth Workshop on Programmability Issues for Heterogeneous Multicores*, 2013.